Buildroot-based Red Pitaya (STEMLab) system development

G. Goavec-Mérou, J.-M Friedt, October 14, 2024

Developing an operating system based on the Linux kernel requires options tailored to the targeted architecture: we will not use, in the proposed approach, a pre-build binary package based distribution, but aim at providing a *consistent framework* for generating manually all tools from sources. The consistent *bootloader*, *toolchain*, operating system *kernel* and filesystem including *binary executables* and *libraries* will be provided by Buildroot [1].

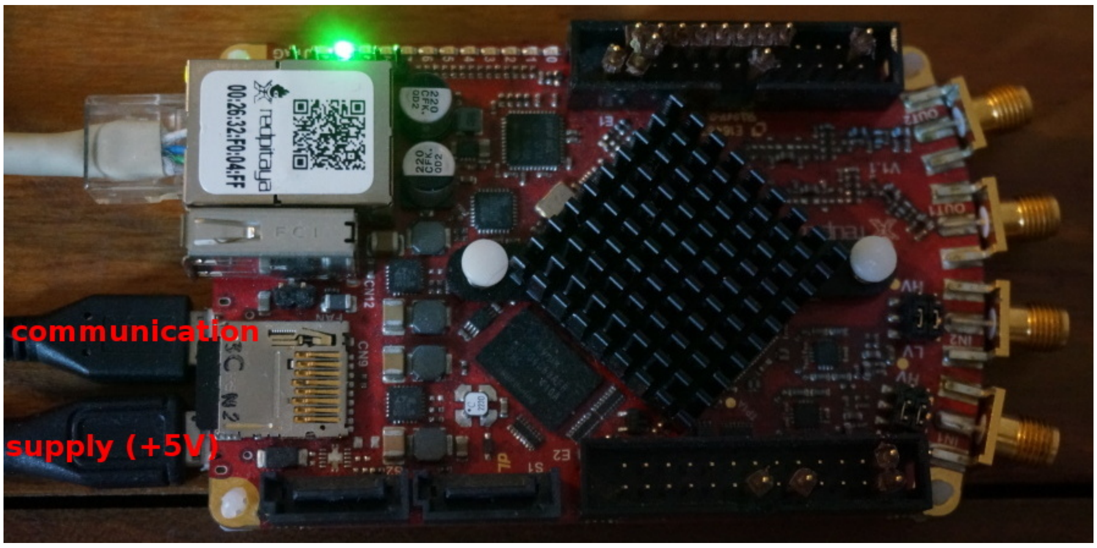


Figure 1: The Red Pitaya board. The USB micro-B connector located closest to the side of the printed circuit board, opposite to the RJ45 Ethernet connector, is used for providing the power supply. The USB micro-B connector located in the middle of the board provides a terminal on a virtual serial port.

1 Prerequisites

All commands prefixed by "#" indicate commands to be executed as administrator ("root"). On some distributions (e.g. Ubuntu), executing as "root" requires prefixing the command with `sudo`, assuming the user is part of the "sudo" group (check in `/etc/group` of the development host computer if the login is documented after the "sudo" group).

All commands prefixed by "$" are to be executed as user. Only execute as few commands as possible as root whose account should only be used during system administration tasks.

Throughout the document the **host** refers to the computer we are working on – most often Intel x86 based architecture, providing a functional GNU/Linux environment through a binary distribution (Debian, Ubuntu ...). The host provides a comfortable working environment as opposed to the embedded **target** whose resources are constrained, usually without graphical interface, and whose performance must be tailored to the application and not the developer needs. Our objective will be to **cross-compile** on the host an efficient operating system and development framework for the target since the latter computer architecture is hardly ever Intel x86 based.

```
# apt-get install build-essential libncurses5-dev u-boot-tools
# apt-get install git libusb-1.0-0-dev pkg-config
```

2 Generating all tools from sources: `buildroot`

Buildroot [1] is an integrated set of tools aimed at

1. compiling a custom toolchain tailored to the targeted system (compilation result in `output/host/usr/bin/`) and most significantly the data bus width (32, 64 bits) and accelerator options (FPU, SIMD instructions),

[1] `https://buildroot.org/`

2. compile a tailored Linux kernel,
3. compile associated GNU tools and libraries.

Several options were available when considering which development framework to use with such a powerful processor as the Zynq fitted on the Red Pitaya: the microprocessor is recent enough that dedicated functionalities of the Linux kernel must be tailored to the targeted plaform, most significantly accessing the FPGA programmable logic (PL) located next to the general purpose dual-processor processing system (PS). Xilinx, manufacturer of the Zynq, has updated a dedicated development branch of the Linux kernel in parallel to the official branch. At the moment, only the Xilinx branch, named `linux-xlnx`, allows to configure the FPGA, using methods which should eventually be integrated into the official kernel. However, the second challenge lies in being able to experiment with the real-time Linux kernel extension – Xenomai – which is only supported by the official kernels. The dilemma hence lies in selecting an official kernel without FPGA support, or the divergent Xilinx kernel without Xenomai support, with an uncertain long term evolution and availability. Hence in this document two toolchain compilations are addressed, one using the Xilinx Linux kernel **with FPGA and Devicetree Overlay (DTBO)** support, and the other one with Xenomai support relying on a mainline kernel but hence **not supporting the dyanamic devicetree update mechanism of the DTBO**.

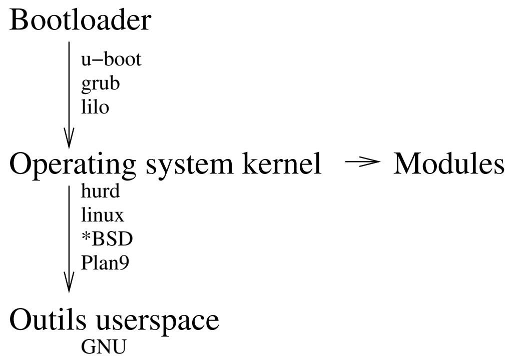


Figure 2: Elements needed for developing and running an operating system: the tools needed to generate all these elements must be consistent.

The advantage of buildroot [2], as well as open-embedded [3], is to provide an integrated and consistent environment that hides a number of details from the developer (Fig. 2). The disadvantage of buildroot, as well as open-embedded, is to provide an integrated environment that hides a certain number of details from the developer (!). We will take advantage of this working environment to compile all the tools needed to work with the Red Pitaya without depending on an external binary distribution, and then to add the Linux real-time extension Xenomai [2]. This extension requires, in addition to an update of the kernel, additional tools in user space that buildroot can manage. Using Xenomai on the Red Pitaya will be described in detail in the section 7.
Buildroot, available at `https://github.com/buildroot/buildroot`, does not support natively the Red Pitaya platform. We must exploit a mechanism for adding such a functionality through the environment variable `BR2_EXTERNAL` which references the absolute directory containing the files missing in the official distribution for the support of a platform. We will therefore download a second repository in addition to the official distribution of `buildroot` in order to add the missing features by pointing to `BR2_EXTERNAL` to a tree structure consistent with the file organization expected by by `buildroot`.

3 Compiling a Linux kernel and image without Xenomai support (no real time support but FPGA configuration through DTBO)

We start from a known official version of `buildroot`, rather than the latest update available on `github`, in order to use a reliable and reproducible version of each tool:

```
wget https://buildroot.org/downloads/buildroot-2024.05.2.tar.gz
```

The default distribution is complemented with additional features for supporting the Red Pitaya:

```
git clone https://github.com/trabucayre/redpitaya.git
```

which cretes a `redpitaya` directoty holding the necessary files. We must tell `buildroot` where these additional files are located: this is achieved with the environment variable `BR2_EXTERNAL`. Either this variable can be defined with the absolute path leading to the `redpitaya` directory, or execute the `sourceme.ggm` script from within the `redpitaya` directory which was just cloned:

```
cd redpitaya
source sourceme.ggm
```

Care must be taken to reload the script in each new terminal, the definition of `BR2_EXTERNAL` being only local to the active terminal and not being propagated to other work sessions.

[2] `buildroot.uclibc.org/`
[3] `www.openembedded.org`

Enter the `buildroot` directory that was just uncompressed and unarchived with

```
tar zxvf buildroot-2024.05.2.tar.gz
```

If you plan on renaming this directory, do it now since after the Buildroot compilation process, some links will be broken if the directory name is changed later. From this directory, configure `buildroot` targeted towards the Red Pitaya:

```
cd buildroot-2024.05.2
make redpitaya_defconfig
```

Then, `make` generates a first functional image by downloading all the necessary sources and compile. At the end of this work which can take from several tens of minutes to hours, and requires 5 to 8 GB of storage space, all Red Pitaya binaries can be found in `output/images`.

⚠ The following lines might **damage any storage medium** if the wrong peripheral is selected. Alway **check** the name of the peripheral associated with the SD card (`dmesg | tail`) before launching the `dd` command.

The image resulting from the compilation sequence is transfered to the SD card with

```
sudo dd if=output/images/sdcard.img of=/dev/sdc
```

in which we have on purpose selected the `/dev/sdc` peripheral since it is hardly ever the appropriate option: most often, the SD card will be called `/dev/sdb` (second hard disk compatible with the Linux SCSI driver) or `/dev/mmcblk0` (case of an internal SD card reader).

⚠ **Warning**: the content of the SD-card, or any mass storage peripheral accessed through the last argument of this command, will be **definitely** lost. Always check twice the name of the target periperal the Buildroot image is written to.

Once the image has been transfered to the SD card, we observe two partitions on the mass storage medium: the first one formated as VFAT (compatible with Microsoft Windows) holding the devicetree description, the Linux kernel, and the bootloader, and the second one holding the GNU/Linux filesystem. This second partition size is exactly the one provided as argument in the Buildroot options `.config` for storing the operating system files, and little space is available for one own additions – we will see later that mainly we wish to copy bitstreams for configuring the FPGA. This lack of space can be corrected by mounting the SD-card and change the second partition size with

1. delete the partition using `fdisk`: lauch the program on the host computer as `root` administrator, display the current configuration (which will be needed below) with `p`, then `d` to erase partition 2,
2. re-create this partitiion but this time extending to the whole space available on the card: `n` to create a new partition, `p` for a primary partition, provide the first sector address as was just displayed earlier (in the screenshot below, this first sector address is 32769), and accept the default selection to use the whole available space. Conclude with `w` to store this new configuration,
3. after leaving `fdisk`, run `resize2fs /dev/sdd2` to resize the filesystem to the size of the partition.

Following all these operations, the GNU/Linux system uses all of the available space, In the following example, we started from a configuration that looked like

```
Device     Boot Start     End Sectors  Size Id Type
/dev/sdd1  *        1   32768   32768   16M  c W95 FAT32 (LBA)
/dev/sdd2       32769 1081344 1048576  512M 83 Linux
```

and after changing the second partition size we end up with

```
Device     Boot Start     End Sectors  Size Id Type
/dev/sdd1  *        1   32768   32768   16M  c W95 FAT32 (LBA)
/dev/sdd2       32769 7796735 7763967  3.7G 83 Linux
```

Asynchronous (RS232-compatible) link emulation over the USB port located in the middle of the printed circuit board is functional: launch `minicom` after connecting a USB-micr B cable, tuned to a 115200 baudrate, 8N1. A prompt stating `redpitaya>` should appear upon hitting the ENTER key.

Various additional features can be added such as a graphical interface server (X11, see appendix B) but are not detailed in the main part of the text but mentioned in various appendices.

3.1 Network configuration

The network configuration between host and target is handled by the interface configuration command `ifconfig` to assign a logic address (Internet Protocol – IP) to a physical interface (defined through its MAC address) and create a point to point link between computers of a sub-network and their gateway. This gateway is in charge of transfering data packets towards computers located outside the sub-network, or even towards the internet if the targeted address is not within the subnet. The command to define the routing table is `route`: we shall not need to specify these routing tables manually in the simple case of the point to point communication of the Red Pitaya with the host computer since GNU/Linux automatically assigns a given subnet to its interface and properly routes packets within the subnet.
The default network configuration is to communicate through the Ethernet interface, with the embedded board IP address given by `ifconfig`.

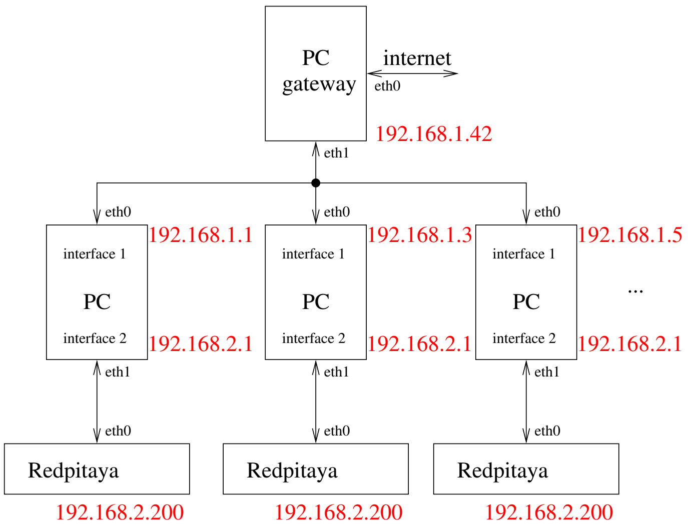


Figure 3: Architecture of a TCP/IP network

We observe that

```
redpitaya> ifconfig
eth0      Link encap:Ethernet  HWaddr 22:3D:C4:D9:DB:CD
          inet addr:192.168.2.200  Bcast:192.168.1.255  Mask:255.255.255.0
...
```

the default IP address is 192.168.2.200, hence a local sub-network [4].

Changing a network interface configuration (on PC or on Red Pitaya) is achieved using the `ifconfig` command. Arguments include the name of the interface and the associated IP address:

```
ifconfig eth0 192.168.2.1
```

Before configuring an interface, the list of all interfaces, whether already configured or not, is obtained with `ifconfig -a`. This command is also used to identify the hardware MAC address as found in the `ether` field: this information is the one transmitted to the gateway when packets are routed towards the internet.

`ifconfig` is slowly becoming obsolete to be replaced with `ip`. The same result as above is achieved with

```
ip address add 192.168.2.1 dev eth0
ip link set dev eth0 up
```

The consistent network configuration between two computers is validated uwing `ping IP_dest` requesting an answer (`ping`) from the remote computer with IP address `IP_dest`.

Once a network configuration is completed and manually validated, it can be automated by writing the configuration in `/etc/network/interfaces`. The rules for configuring the Red Pitaya IP configuration can be added as follows:

```
auto eth0
iface eth0 inet static
  address 192.168.2.200
  netmask 255.255.255.0
  gateway 192.168.2.1
```

This configuration is activated on the Red Pitaya with `ifup eth0`. If the interface was already configured, it might be needed to cancel this previous configuration using `ifdown eth0`. If such a configuration is also used on the PC, make sure the Network Manager is not conflicting with `/etc/network/interfaces`: Network Manager can be told not to

[4] the address ranges for private local networks, which are never routed towards the internet, are described in RFC1918 available at `https://tools.ietf.org/html/rfc1918`

mess with interfaces defined in `/etc/network/interfaces` by stating `managed=false` in `/etc/NetworkManager/NetworkMana`
of the PC Linux distribution.

We shall later need a web server: `lighttpd` is a light and stable option. Adding a new application with Buildroot is achieved by running `make menuconfig` and search for the location where the web server can be activated, possibly by searching (`/` followed by `lightt`) a keyword, or by activating directly `Target packages` → `Networking applications` → `lighttpd`.

Life will be easier by installing a text editor on the target system. The most efficient, `vim`, requires activating the `wchar` and `curses` of the toolchain with the sequence `Toolchain` → `Enable WCHAR support` and `Target packages` → `Libraries` → `Text and terminal handling` → `ncurses`. Once these libraries have been activated, `vim` support in `busybox` is activated by `Target packages` → `Show packages that are also provided by busybox` and `Target packages` → `Text editors and viewers` → `vim`. Many other alternative text editors are also available in the same menu. Once selected, quit `make menuconfig`, run `make` and transfer the resulting image to the SD card upon completion of the compilation.

The addition of the new features is validated by launching the Red Pitaya and making sure the new software is indeed available:

```
redpitaya> which lighttpd
/usr/sbin/lighttpd
```

3.1.1 Domain name correspondence with IP addresses

The domain name server, in charge of translating the domain name to an IP address – DNS – is defined in `/etc/resolv.conf`. This information is needed only for reaching domains on the internet, for example `free.fr` or `google.com`, but is useless as long as we are restricted to local connections in which the targeted system is identified with its IP address. The DNS information is already filled on the PCs, but the Red Pitaya is unaware of the DNS IP address to relate a domain name, e.g. `sequanux.org`, and its IP address (188.165.36.56). Filling the configuration file
`/etc/resolv.conf` with information such as

```
nameserver 194.57.91.200
nameserver 194.57.91.201
```

with 194.47.91.20{0,1} the domain name servers of Université de Franche Comté will solve the issue. On a differente site, the DNS address will be found by looking into the `/etc/resolv.conf` of a local PC. A generic solution is to selected `9.9.9.9` as DNS[5]. server address

When connecting using the Secure Shell `ssh` from a PC to the Red Pitaya, the only administrator account `root` password must be provided.

The default `root` password is **root**.

3.1.2 NFS (*Network File System*)

In the case of room 235B, a directory has been allowed on each PC to be shared with the Red Pitaya: this directory is `/home/etudiant/nfs`. This directory represents the more generic reference to `/home/utilisateur/target` in the general description that follows.

Sharing a directory by NFS allows for making the user believe that part of the (host) PC hard disk drive can be accessed as a directory on the (target) embedded Red Pitaya board. Sharing virtually files through such a network connection provides a flexible working environment since after cross-compiling a binary file on the (fast) PC, the resulting executable is immediately available in the shared directory for executing on the embedded board.

Achieving this result requires, on the PC, to create a `/home/utilisateur/target` directory which will be the shared directory. Then in the `/etc/exports` configuration file of the host (running the NFS server), we allow sharing this directory by adding `/home/utilisateur/target *(rw,sync)`

Restart the NFS server on the PC to comply with the new rule:

```
#/etc/init.d/nfs-kernel-server restart
```

[5] `https://www.quad9.net/`

This directory will allow sharing files between the host and the target, for examples executables cross-compiled on the PC and we wish to execute on the Red Pitaya, or in the other direction data that were collected on the Red Pitaya to be processed on the host using a graphical interface such as `gnuplot` or GNU/Octave for displaying the charts. For accessing the directory on the Red Pitaya, the remote directory is mounted using `mount -o nolock IP_PC:/home/utilisateur/target /mnt`: the content of `/home/utilisateur/target` on the host is now accessible in the `/mnt` directory of the target.

Connecting the Red Pitaya to the internet by configuring the PC as gateway (optional)

We only briefly mention the ability to use the PC as a gateway that translates the Red Pitaya IP requests and broadcast towards the targeted server on the internet: this capability will not be used at first and can be skipped if not needed. The tool for achieving such address translation is called `iptables`: a configuration example for transfering all packets between `eth0` and `eth1` interfaces of the host PC is provided below. This configuation only makes sense on the PC where we assume that `eth0` is connected to the subnetwork linked to the gateway allowing to reach the internet, and `eth1` is connected to the subnetwork holding the la Red Pitaya. These commands all require administration (`root`) credentials.

```
monpath=/sbin
echo "1" > /proc/sys/net/ipv4/ip_forward
echo "1" > /proc/sys/net/ipv4/ip_dynaddr
${monpath}/iptables -P INPUT ACCEPT
${monpath}/iptables -F INPUT
${monpath}/iptables -P OUTPUT ACCEPT
${monpath}/iptables -F OUTPUT
${monpath}/iptables -P FORWARD ACCEPT
${monpath}/iptables -F FORWARD
${monpath}/iptables -t nat -F

${monpath}/iptables -A FORWARD -i eth0 -o eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT
${monpath}/iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
${monpath}/iptables -A FORWARD -j LOG

${monpath}/iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

3.2 Executing in an emulator

Lacking hardware as is often the case for software developers of embedded systems, the image resulting from `buildroot` compilation can be executed in an emulator, here `qemu` emulating an ARM processor. We will use at first, as long as the `eth0` network interface is not needed, the Debian GNU/Linux package version provided as `qemu-system-arm`. Once this packet has been installed, go to the Buildroot `output/images` directory and run

```
qemu-system-arm -append "console=ttyPS0,115200 root=/dev/mmcblk0 rw earlyprintk" -M xilinx-zynq-a9 \
-m 1024  -serial mon:stdio -dtb zynq-red_pitaya.dtb -nographic -kernel zImage  -sd rootfs.ext4
```

to boot Linux, load the (`rootfs`) filesystem and complete loading the system until a prompt appears. All system commands are functional, but accessing hardware requires of course that such peripherals are specifically emulated, a functionality which `qemu` might not provide. For example, GPIO related registers seem to be properly emulated

```
redpitaya> echo "908" > /sys/class/gpio/export
redpitaya> echo "out"  > /sys/class/gpio/gpio908/direction
redpitaya> echo "1"  > /sys/class/gpio/gpio908/value
```

even though the consequences of writing to a peripheral controling register are not obvious.

The drawbacks of this approach are that 1/ the Ethernet interface of the Red Pitaya is not emulated and 2/ only the guest operating system on the Red Pitaya emulated with `qemu` can access the host operating system on the PC running `qemu`, but connections from the host to the guest are are not supported as long as dedicated network interface has not been created. In order to compensate for these limitations, the specific branch of `qemu` provided by Xilinx as described at `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/821395464/QEMU+User+Documentation` is compiled, and most significantly with the support of the Red Pitaya Zynq processor as detailed at `https://`

xilinx-wiki.atlassian.net/wiki/spaces/A/pages/189530183/Zynq-7000. Notice that compiling all of qemu is not needed but that after adding the devicetree compiler with `git submodule update --init dtc` we will only compile the wanted target, namely `$./configure --target-list="aarch64-softmmu" --enable-fdt --disable-kvm --d`

Thanks to the compilation of the binary resulting from the archive provided by Xilinx at https://github.com/Xilinx/qemu as described below, we end up with `aarch64-softmmu/qemu-system-aarch64` with the support of the `arm-generic-fdt-7series` machine whose Ethernet interface is detected as `eth0`. The command line to launch `qemu` with such a functionality is

```
sudo .../aarch64-softmmu/qemu-system-aarch64 -append "console=ttyPS0,115200 root=/dev/mmcblk0 \
rw earlyprintk" -M arm-generic-fdt-7series -machine linux=on -smp 2 -m 1024  -serial mon:stdio \
-dtb zynq-red_pitaya.dtb -nographic -kernel zImage  -sd rootfs.ext4  -net nic -net nic -net nic \
-net nic -net tap,downscript=no
```

with the whole command launched from Buildroot's `output/images` directory to avoid filling the full path of all the files (devicetree `zynq-red_pitaya.dtb`, root filesystem `rootfs.ext4`, Linux kernel `zImage`). Notice that this command must be launched as **administator (root) on the PC** in order to be allowed to create the `tap0` network interface on the host.

Once the virtual Red Pitaya launched in `qemu-system-aarch64`, we observe with (`ifconfig -a`) that the `eth0` network interface on the target is available. On the PC, a new interface was created when launching `qemu` named `tap0`: on the PC, `sudo ifconfig -a | grep tap` displays the properties of this new interface. **Beware**: if an older execution of qemu was improperly killed or is running as a background task, a new `tap` interface such as `tap1` might be created, Make sure to properly identify which interface is connected to which running copy of `qemu`.

The `tap` interface having been identified, we must still configure an address on the PC in the same subnet – but of course with a different least significant byte – than the Red Pitaya address. Once this configuration is completed on the host and the target, a successful `ping` validates the proper operation of the link.

Figure 4 demontrates the configuration of the `eth0` interface when emulating the Red Pitaya (address 192.168.0.10) in `qemu`, consistent with the `tap0` interface created and configured on the host (same sub-net 192.168.0.x with the PC arbitrarily assigned the 192.168.0.5 address) and the ability of a web client running on the host to access the `lighttpd` server executed on the guest.

A last advice: if we wish to avoid the hassle of secure shell connexion as provided by `ssh` between the host and the guest, a telnet server `telnetd` is available with `buildroot` for the Red Pitaya as a feature provided by busybox accessed with `make busybox-menuconfig`→`networking`→`telnetd`. With this server, the Red Pitaya becomes accessible with `telnet 192.168.0.10` (according to the configuration visible on Fig. 4) without performance degradation due to sharing public cryptographic keys, but with information circulating un-encrypted over the network.

Files can be transfered from host to target using `scp`. Furthermore, NFS is also functional in this framework, as long as a functional TCP/IP link exists between host and guest and that the host directory to be shared has been defined by including the guest IP address in `/etc/exports` (see previously section 3.1.2).

Leaving `qemu` cleanly while resetting all variable and memory allocations is achieved similar `minicom`, with "CTRL-a" followed by "x".

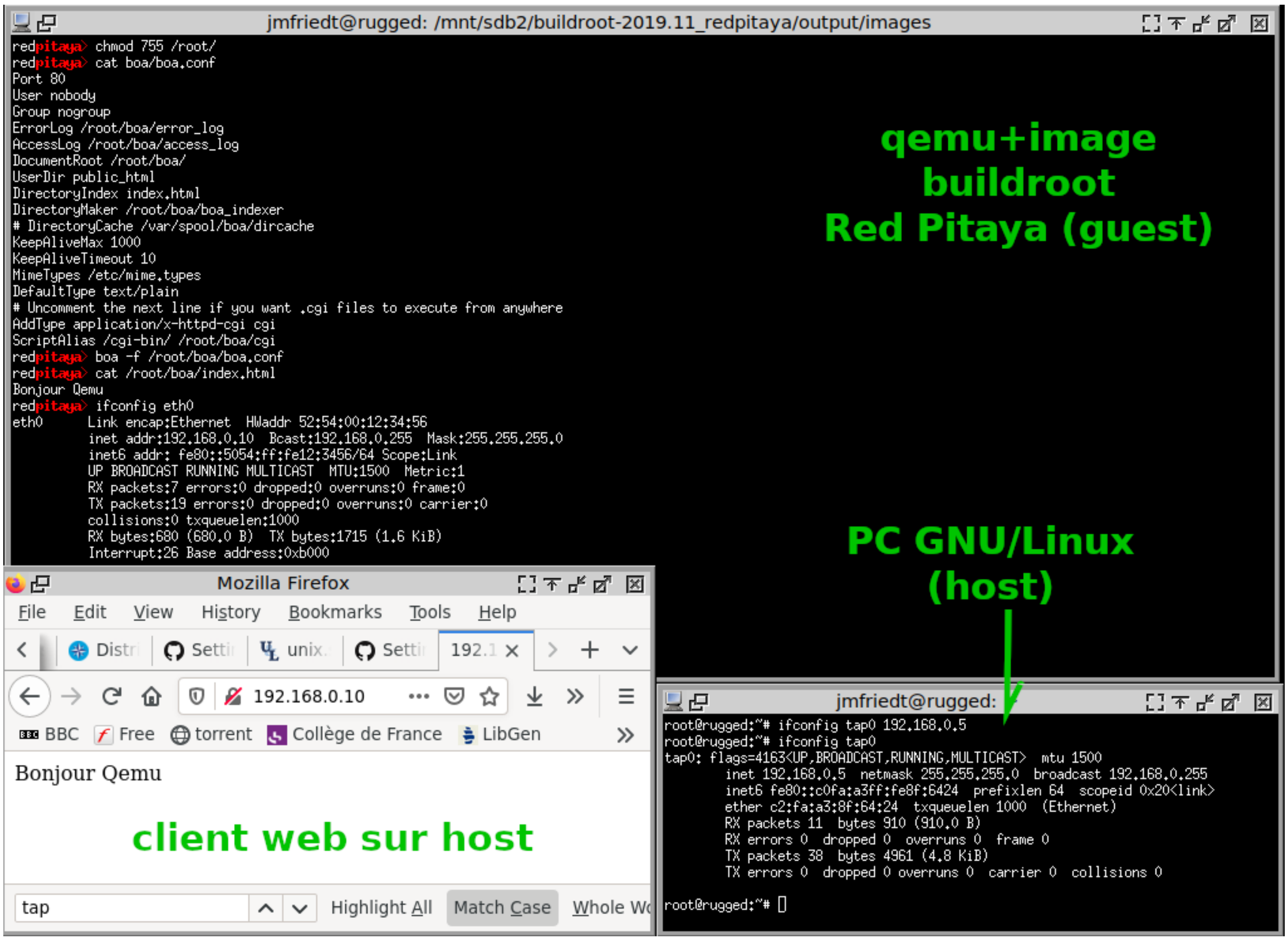
3.3 Web server

We have seen (section 3.1) how to complement the Redpitaya Buildroot configuration in order to add support for a light web server, implementing CGI script execution, named `lighttpd`. A default configuration is found in `/etc/lighttpd/lighttpd.conf` using the `/var/www/` directory for storing the files accessible through a web connection. The server is launched as a daemon using `lighttpd -f /etc/lighttpd/lighttpd.conf`. If an error occurs upon launching `lighttpd`, check the error log located in `/var/log/lighttpd-error.log` that the issue is not related to "pcre support is missing for: dir-listing.exclude". In case this error occurs, we might deactivate the faulty module by commenting out

```
# include "conf.d/dirlisting.conf"
```

in `/etc/lighttpd/lighttpd.conf`.

A basic static web page allows for testing proper operation of the web server and connectivity to the web browser on the host computer: on the Red Pitaya, edit a file named `index.html` in the `/var/www` directory including the sentence `Hello World` and check that it is displayed when typing the Red Pitaya IP address in the host web browser.



```
redpitaya> chmod 755 /root/
redpitaya> cat boa/boa.conf
Port 80
User nobody
Group nogroup
ErrorLog /root/boa/error_log
AccessLog /root/boa/access_log
DocumentRoot /root/boa/
UserDir public_html
DirectoryIndex index.html
DirectoryMaker /root/boa/boa_indexer
# DirectoryCache /var/spool/boa/dircache
KeepAliveMax 1000
KeepAliveTimeout 10
MimeTypes /etc/mime.types
DefaultType text/plain
# Uncomment the next line if you want .cgi files to execute from anywhere
AddType application/x-httpd-cgi cgi
ScriptAlias /cgi-bin/ /root/boa/cgi
redpitaya> boa -f /root/boa/boa.conf
redpitaya> cat /root/boa/index.html
Bonjour Qemu
redpitaya> ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:56
          inet addr:192.168.0.10  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::5054:ff:fe12:3456/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:680 (680.0 B)  TX bytes:1715 (1.6 KiB)
          Interrupt:26 Base address:0xb000
```

```
root@rugged:~# ifconfig tap0 192.168.0.5
root@rugged:~# ifconfig tap0
tap0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.5  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::c0fa:a3ff:fe8f:6424  prefixlen 64  scopeid 0x20<link>
        ether c2:fa:a3:8f:64:24  txqueuelen 1000  (Ethernet)
        RX packets 11  bytes 910 (910.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38  bytes 4961 (4.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@rugged:~# 
```

Figure 4: Top-left the guest: Xilinx `qemu` emulating the Red Pitaya. Bottom-left the web client executed on the host and connecter to the web server running on the guest. Bottom-right the host network configuration created when launching `qemu` with the `-net tap` option.

Further interaction with hardware will require the ability to dynamically analyze the URL arguments and act accordingly, hence launching a script on the Red Pitaya when requested by the host web browser. Indeed, CGI (*Common Gateway Interface*) allows for dyanamically generating web pages by executing a script generating HTML sentences that can be read by the client. The protocol specification, `http://www.ietf.org/rfc/rfc3875.txt`, indicates the means for sharing information between client and server.

CGI script support is added by storing in `/etc/lighttpd/conf.d/cgi.conf` a configuration file with the following content

```
server.modules += ( "mod_cgi" )
cgi.assign                 = ( ".pl"  => "/usr/bin/perl",
                               ".cgi" => "/bin/sh",
                               ".py"  => "/usr/bin/python",
                               ".php" => "/usr/bin/php-cgi" )

index-file.names           += ( "index.pl",   "default.pl",
                               "index.rb",   "default.rb",
                               "index.erb",  "default.erb",
                               "index.py",   "default.py",
                               "index.php",  "default.php" )
```

and adding **at the end** of `/etc/lighttpd/lighttpd.conf` the line `include "conf.d/cgi.conf"` for loading this CGI configuration file.

Once the update is completed, the web server is restarted by using `/etc/init.d/S50lighttpd restart`, possibly after killing the previous process that was launched manually by finding its identifier with the `ps` command.

⚠ Never run services with administration privileges (`root`). Even if such a configuration makes accessing hardware resources much easier, it break unix security rules based on the clear separation between services (examples of poor web server configration exploits: *Exploiting Network Surveillance Cameras Like a Hollywood Hacker*, Black Hat 2013, at `https://www.youtube.com/watch?v=B8DjTcANBx0`). Here, the web server is run as user and group `www-data` which have no administration rights so that a security failure in the web server would not lead to a security breach in the whole system.

For executing CGI scripts, we might create a sub-directory `/var/www/cgi` (in agreement with the configuration file) and store there the shell script (since `.cgi` extensions are associated with bash scripts):

```
#!/bin/sh
echo -e "Content-type: text/html\r\n\r\n"
echo "Hello CGI"
```

which is launched from the web client by providing the url `http://IP/cgi/script.cgi`. We will of course remember to make the script executable (`chmod 755 script.cgi` in the `cgi` directory). This method hence allows for dynamically generating web pages, for example for displaying sensor reading measurements.

The script running on the server can, for example, read a sensor or GPIO state and update the web page accordingly. More interesting, the HTML request can include variable names and their values to be used by the script. The two methods meeting this objective are POST and GET. The latter will be used here since the data are shared between client and server through the url, easier to implement client et serveur [6]. Hence, providing arguments to the CGI script is achieved with URLs stating

```
http://192.168.2.200/cgi/scrpit.cgi?variable1=val1&variable2=val2&var3=val
```

The only subtely in this usage is

1. understand the execution context of the CGI script, whatever language it is written in (the first line, as in all scripts, informs on the location of the interpreter),
2. fetch the list of variables provided through the url. The variables including these parameters are described at page 8 of `http://www.ietf.org/rfc/rfc3875.txt`.

Before we can interact with hardware from a CGI script, we must learn how to interact with GPIOs from the interactive shell on the Red Pitaya.

3.4 Accessing hardware from the shell

If we wish to use the LED control driver: the `/sys/class/leds` virtual directory provides access to the associated kernel driver for controlling the LEDs. The following sequence

```
redpitaya> cd /sys/class/leds/
redpitaya> echo "1" > led8/brightness   # orange
redpitaya> echo "0" > led8/brightness
redpitaya> echo "0" > led9/brightness   # rouge
redpitaya> echo "1" > led9/brightness
```

allows for checking proper hardware resource control by the Linux kernel.

The `leds` interface is more limited than the general `gpio` interface but easier to use. Various interfaces are provided to the user to access hardware resources. The current trend is to expose these interfaces through the `/sys` directory. Two methods are provided for accessing GPIOs connected to the processor (MIO in the Zynq naming convention, as opposed to EMIO connected to the PL) thanks to the Linux drivers: `/sys/class/leds` and `/sys/class/gpio`.

A complementary approach is to use `/sys/class/gpio` [7] which provides lower access to the GPIO since LEDs are a sub-set of GPIOs: the LED driver relies on the GPIO driver.

If drivers are compiled as modules, we observe that they are loaded with

[6] notice that the apparent protection provided by the POST post method against a malicious usage is trivially bypassed with tools such as `curl`

[7] `https://www.thegoodpenguin.co.uk/blog/stop-using-sys-class-gpio-its-deprecated/` claims that `/sys/class/gpio` is becoming obsolete, but remains available in the 6.6 kernel so far ...

```
redpitaya> lsmod
Module                  Size  Used by    Not tainted
gpio_zynq               6509  2
leds_gpio               2890  0
led_class               3299  1 leds_gpio
```

For accessing GPIOs, a driver implements `/sys/class/gpio` [8]. One feature of `/sys` is to communicate through human readable sentences (as opposed to binary valus shared with peripherals accessed through `/dev`).

Kernel versions < 6 (4., 5.x)

Practically, the `gpiolib` drivers assumes that a digital port exposes at most 32 input-output pins. Hence, the index of pin `n` on port `P` is `P`×32+`n`, with port A indexed as 0, B indexed as 1 and so on. In the case of the Red Pitaya, pins refered to as MIO are indexed starting with the value 906 in kernels prior to 6, and with values starting with 512 for kernels 6.x. Hence, MIO0 is indexed as 906 on pre-6 kernel and 512 on the 6.x kernel, and MIO7 is indexed as 913 on pre-6 kernel and as 519 on 6.x kernels: the two LEDs – orange and red – are connected to these two pins. We demonstrate control of these peripherals with the following sequence.

In a first step, we request control of th resource with

```
redpitaya> echo "906" > /sys/class/gpio/export # for kernel < 6, 512+ otherwise
-sh: write error: Device or resource busy
```

which fails since the LED driver already controls these resources, emphasizing the role of the kernel in maintaining consistency when accessing a given resource by multiple users.

Since kernel 6 (check with `uname -a` which kernel version you are running), the GPIO indices have been renamed as can be checked with the `debugfs` available when compiling the Linux kernel with `DEBUG_FS=y` and `mount -t debugfs none /sys/kernel/debug/` so that `cat /sys/kernel/debug/gpio` displays

```
gpiochip0: GPIOs 512-629, parent: platform/e000a000.gpio, zynq_gpio:
 gpio-512 (                    |led8                ) out lo
 gpio-519 (                    |led9                ) out lo
```

As in the previous discussion, but replacing 906 or 913 with 512 and 519 respectively, we can access the GPIO with

```
echo 512 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio512/direction
echo 1 > /sys/class/gpio/gpio512/value
```

only once the LED driver has freed the resources.

We have taken care to configure the drivers handling hardware resources as modules, so that these resources can be freed by removing the drivers from memory:

```
redpitaya> rmmod leds_gpio
redpitaya> rmmod led_class
redpitaya> echo "906" > /sys/class/gpio/export  # for pre-6 kernel, 512 otherwise
```

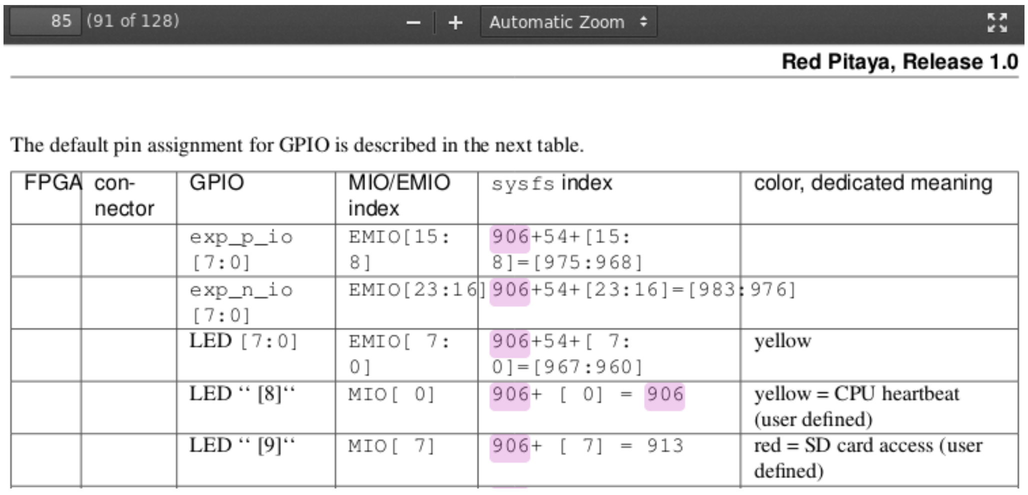
for pre-6 kernels, or for 6.x kernels:

```
redpitaya> rmmod leds_gpio
redpitaya> rmmod led_class
redpitaya> echo "512" > /sys/class/gpio/export
```

Now, the previous request succeeds – no error message is given – and leads to the creation of a new directory holding the pseudo-files for accessing the hardware resources through requests to the kernel: `gpio906` for pre-6 kernels and `gpio512` for 6.x kernels, [9]. These pseudo-files handle the communication direction (`out` or `in`) and, for pins configured as outputs, their state (`1` or `0`).

[8] make sure to activate such functionalities in the kernel configuration with `Device Drivers`→`GPIO Support`→`/sys/class/gpio/... (sysfs interface)`

[9] `https://forum.redpitaya.com/viewtopic.php?t=1158` explains that LED8 = MIO[0]=gpio906 and LED9 = MIO[7]=gpio913



Red Pitaya, Release 1.0

The default pin assignment for GPIO is described in the next table.

FPGA	con- nector	GPIO	MIO/EMIO index	sysfs index	color, dedicated meaning
		exp_p_io [7:0]	EMIO[15: 8]	906+54+[15: 8]=[975:968]	
		exp_n_io [7:0]	EMIO[23:16]	906+54+[23:16]=[983:976]	
		LED [7:0]	EMIO[7: 0]	906+54+[7: 0]=[967:960]	yellow
		LED “ [8]“	MIO[0]	906+ [0] = 906	yellow = CPU heartbeat (user defined)
		LED “ [9]“	MIO[7]	906+ [7] = 913	red = SD card access (user defined)

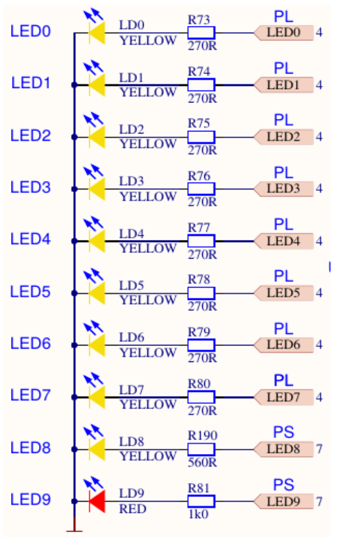


Figure 5: LEDs available on the Red Pitaya: notice that only LED8 and LED9 are accessible from the PS (i.e. without configuring the FPGA of the PL). Figures taken from `media.readthedocs.org/pdf/rpdocs/latest/rpdocs.pdf` and `downloads.redpitaya.com/doc/Red_Pitaya_Schematics_STEM_125-10_V1.0.pdf` respectively
.

```
redpitaya> echo "out" > /sys/class/gpio/gpio906/direction
redpitaya> echo "1" > /sys/class/gpio/gpio906/value
redpitaya> echo "0" > /sys/class/gpio/gpio906/value
```

for pre-6 kernels and

```
redpitaya> echo "out" > /sys/class/gpio/gpio512/direction
redpitaya> echo "1" > /sys/class/gpio/gpio512/value
redpitaya> echo "0" > /sys/class/gpio/gpio512/value
```

for 6.x kernels, validates that accessing hardware resources from the shell is functional.

3.5 Back to lighttpd and CGI scripts

Now that we understand access to the GPIO and LED hardware from the shell, this knowledge is transposed to the CGI script. Be aware though that the main challenge will be ownership of the resources and being allowed to access hardware as non-administator user.

A basic example of a functional CGI script is provided below, which requires to understand the last line to refer to the section concerning hardware access at 3.4, as

```
[root@buildroot cgi]# less cgi/script.cgi
#!/bin/sh
echo -e "Content-type: text/html\r\n\r\n"
echo "Hello CGI<br>"
echo $QUERY_STRING "<br>"
led=‘echo $QUERY_STRING | cut -d= -f2‘
echo $led "<br>"
echo $led > /sys/class/leds/led8/brightness
```

located in `/var/www/cgi`. The string provided as argument in the URL is accessed through the variable `$QUERY_STRING` which is here split in its individual items by using the `cut` command.

We must however make sure that the owner of the web server, defined as the user `www-data` for `lighttpd` to isolate permissions and limit the risks of malicious access in case security of the server is breached, has access to the requested hardware resources. That is usually not the case and we must manually update permissions:

```
# ls -l /sys/class/leds/led*/*
lrwxrwxrwx    1 root     root             0 Jan  1 00:13 brightness
...
# chown www-data.www-data /sys/class/leds/led*/brightness
# ls -l /sys/class/leds/led*/
```

for `lighttpd`.

Demonstrate how you can define the state of *both* LEDs 8 and 9 by providing a GET url to `lighttpd` with both arguments.

3.6 Concept of `devicetree`

A SOC (*System On Chip*) provides many peripherals around a same processing core. Describing these peripherals can become complex and require inheriting properties: on ARM architectures, the most recent approach lies in using the `devicetree` as was already done on SPARC or PowerPC architectures. This file, with the extension `dts` to be readable by a programmer, is compiled as a `dtb` binary file readable by kernel modules and drivers.

Accessing LEDs through the `/sys/class/leds` interface configured by the `devicetree` is achieved by declaring which pin is associated with the functionality, leading to the creation of a directory holding the pseudo-files for handling pin states.

We observe the configuration of MIO0 and MIO7 as interfaces led8 and led9 respectively in the devicetree section found in `redpitaya/board/redpitaya/patches/linux/xilinx-v2024.1/0002-redpitaya12-add_devicetree.patch` of BR2_EXT
defining the Red Pitaya peripherals:

```
gpio-leds {
 compatible = "gpio-leds";
 led-8-yellow {
  label = "led8";
  gpios = <&gpio0 0 0>;
  default-state = "off";
  linux,default-trigger = "mmc0";
 };
 led-9-red {
  label = "led9";
  gpios = <&gpio0 7 0>;
  default-state = "off";
  linux,default-trigger = "heartbeat";
 };
};
```

The *devicetree* will be our most trusted interface when defining specific platform configurations to the Linux kernel, as will be described later in detail, especially when configuring the programmable logic (PL) of the chip.

4 TCP/IP and UDP/IP servers using a C program

While Python3 will provide compact and easy means for implementing a server on the Red Pitaya, the framework is complex to install and not readily available in our Buildroot image. We shall hence focus on a C implementation of a TCP and UDP server over IP and check how they interact from the host using `telnet` for accessing the TCP server or `netcat` for accessing the UDP server.

```c
#include <sys/socket.h>
#include <resolv.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>

#define MY_PORT         9999
#define MAXBUF          1024

int main()
{int sockfd;
 struct sockaddr_in self;
 char buffer[MAXBUF];
 // socket type (AF = IPv4, STREAM=TCP)
 sockfd = socket(AF_INET, SOCK_STREAM, 0);
 bzero(&self, sizeof(self));
 self.sin_family = AF_INET;
 self.sin_port = htons(MY_PORT);
 self.sin_addr.s_addr = INADDR_ANY;
 bind(sockfd, (struct sockaddr*)&self, sizeof(self));
 listen(sockfd, 20);                        // wait for incoming connection
 while (1)
 {struct sockaddr_in client_addr;
  int mysize,clientfd;
  unsigned int addrlen=sizeof(client_addr);
  clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
  printf("%s:%d connected\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
  mysize=recv(clientfd, buffer, MAXBUF, 0);
  send(clientfd, buffer, mysize, 0);
  close(clientfd);
 }
 close(sockfd);return(0);  // Clean up (should never get here)
}
```

implements the TCP server sequence

1. socket (protocol)
2. bind (port)
3. listen (blocking wait)
4. accept
5. send/recv
6. close

Be aware of the endianness of both speakers when sharing multi-byte data structures: `hton` (host to network) and `ntoh` (network to host) followed by `s` for short (two bytes) or `l` (long for 4 bytes) make sure the endianness between both speakers is consistent: the internet is by convention **big-endian**, while most processor architectures (x86, most ARM, RISC-V) are little-endian, hence requiring the conversion.

For a UDP server:

```
#include <sys/socket.h>
#include <resolv.h>
#include <arpa/inet.h>

#define BUFSIZE         1024

void alltoupper(char* s)
{while ( *s != 0 ) *s++ = toupper(*s);}

int main()
{ char buffer[BUFSIZE];
  struct sockaddr_in addr;
  int sd, addr_size, bytes_read;

  sd = socket(PF_INET, SOCK_DGRAM, 0);
  addr.sin_family = AF_INET;
  addr.sin_port = htons(9999);
  addr.sin_addr.s_addr = INADDR_ANY;
  bind(sd, (struct sockaddr*)&addr, sizeof(addr));
  do {bzero(buffer, BUFSIZE);addr_size = BUFSIZE;
      bytes_read=recvfrom(sd,buffer,BUFSIZE,0, \
         (struct sockaddr*)&addr,&addr_size);
      printf("Connect: %s:%d %s\n",inet_ntoa(addr.sin_addr),\
         ntohs(addr.sin_port), buffer);
      alltoupper(buffer);
      sendto(sd,buffer,bytes_read,0,(struct sockaddr*)&addr, \
         addr_size);
  } while ( bytes_read > 0 );
  close(sd);return 0;
}
```

Notice how the `socket()` argument was switched from `SOCK_STREAM` to `SOCK_DGRAM`. For testing, `netcat` is used with the `-u` argument for UDP followed by the IP and port: `echo "toto" | nc -u IP 9999`.

Watch using `tcpdump -i eth1 -XXX` how the packets transfered from PC to Red Pitaya can be seen on the interface

Share data encoded on more than 1-byte and observe the endianness issue of the byte organization between sender, network and receiver

5 Hardware access using a C program

Accesing hardware resources from the shell is functional, but 1/ is slow since it must reach through all the abstraction layer between userspace and the hardware including the kernel and 2/ assumes that someone before us has already implemented the driver to control the hardware. We will here discuss how to control hardware without relying on the kernel – an efficient means for quickly prototyping as we are used to when working with microcontrollers not running an operating system but against the principles when developing with GNU/Linux – which will later be useful when qualifying the latencies introduced by the operating system.

`buildroot` has provided a consistent development framework and most significantly a cross-compiler in `output/host/bin` named `arm-linux-gcc`. The datasheet describing the Zynq registers is found at
`https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. As usual, we find there all the addresses of the registers used to interact with the hardware as well as their usage. As usual, such a complex datasheet hides the details that make the difference between a functional and a failing program. Here, the surprise lies, as was the case with the STM32, on the **need to activate the clock driving the GPIO peripheral**: if not activated, reading or writing to a register associated with this peripheral will lead to a null result (0x00). The solution is provided, for an MIO, at `https://support.xilinx.com/s/question/0D52E00006hpU2mSAE/using-genericuio-with-ps-gpio` which explains which clock to activate (bit 22 of the APER_CLK_CTRL register). Furthermore, remember that a `unsigned int *h;` pointer is incremented with 4-byte steps, so that `h+4` points towards the address of `h` incremented by **16 octets** et not 4 bytes as would be erroneously expected. Thus, we find easier to define `unsigned char *h;` which avoids any surprise when incrementing addresses.

The example below allows for blinking the two LEDs connected to the two MIOs driven by the processing system.

```c
// arm-linux-gcc -Wall -o led led.c

#include <stdio.h>
#include <fcntl.h>    // O_*
#include <unistd.h>   // close
#include <stdlib.h>   // atoi
#include <stdint.h>   // uint32_t
#include <sys/mman.h> // mmap

//#define TYPE int  // si on definit h comme int*, alors l'increment est en multiple de 4 !
#define TYPE char

int main(int argc, char** argv)
{ const int base_addr1= 0xF8000000;
  const int base_addr2= 0xe000a000;
  TYPE *h    = NULL; // int *h = NULL;
  int map_file        = 0;
  unsigned int page_addr,page_offset;
  unsigned page_size=sysconf(_SC_PAGESIZE);

  uint32_t value= 0x80; // 1 ou 128 pour LED orange ou rouge
  if (argc >= 2) value = atoi(argv[1]);
  printf("value = 0x%x\n",value);

  page_addr   = base_addr1 & (~(page_size-1));
  page_offset= base_addr1-page_addr;
  printf("page=0x%x size=0x%x offset=0x%x\n",page_addr,page_size,page_offset);

  // mmap the device into memory
  map_file = open("/dev/mem", O_RDWR | O_SYNC);
  h=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,map_file,page_addr);
  if (h<0) {printf("mmap pointer: %x\n",(int)(h));return(-1);}

  // exemple baremetal pour debloquer acces SLCR registers
  // https://forums.xilinx.com/xlnx/attachments/xlnx/EDK/29780/1/helloworld.c
  printf("@      -> %x %x \n",(unsigned int)h,(unsigned int)(h+0x0c));
  printf("init   -> %x \n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));
  *(unsigned int*)(h+0x04/sizeof(TYPE))=0x767b; // lock
  printf("lock   -> %x\n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));
  *(unsigned int*)(h+0x08/sizeof(TYPE))=0xDF0D; // unlock
  printf("unlock -> %x\n",(*(unsigned int*)(h+0x0c/sizeof(TYPE))));

  // IL FAUT L'HORLOGE DE GPIO ! bit 22 de 0xF8000000+0x0000012C, sinon meme lecture echoue (0x00)
  // https://forums.xilinx.com/t5/Embedded-Linux/Zynq-mmap-GPIO/td-p/368601
  printf("ck -> %x\n",(*(unsigned int*)(h+0x12c/sizeof(TYPE))));
  *(unsigned int*)(h+0x12c/sizeof(TYPE))|=(1<<22);
  printf("ck -> %x\n",(*(unsigned int*)(h+0x12c/sizeof(TYPE))));

  page_addr = (base_addr2 & (~(page_size-1)));
  page_offset = base_addr2 - page_addr;
  printf("%x %x\n",page_addr,page_size);
  h=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,map_file,page_addr);
  if (h<0) {printf("mmap pointer: %x\n",(int)(h));return(-1);}
  //write_reg(0xE000A000, 0x00000000, 0x7C020000); //MIO pin 9 value update
  //write_reg(0xE000A000, 0x00000204, 0x200);  //set direction of MIO9
  //write_reg(0xE000A000, 0x00000208, 0x200);  //output enable of MIO9
  //write_reg(0xE000A000, 0x00000040, 0x00000000); //output 0 on MIO9
  //data = read_reg(0xE000A000, 0x00000060);       //read data on MIO

  // https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf p.1347
  *(unsigned int*)(h+page_offset+0)=(value);     // GPIO programming sequence :p.386
  printf("hk: %x\n",*(unsigned int*)(h+page_offset));
  *(unsigned int*)(h+page_offset+0x204/sizeof(TYPE))=value;  // direction
  printf("hk+204: %x\n",*(unsigned int*)(h+page_offset+(0x204)/sizeof(TYPE)));
  *(unsigned int*)(h+page_offset+0x208/sizeof(TYPE))=value;  // output enable
  printf("hk+208: %x\n",*(unsigned int*)(h+page_offset+(0x208)/sizeof(TYPE)));
  *(unsigned int*)(h+page_offset+0x040/sizeof(TYPE))=value;  // output value
  printf("hk+40: %x\n",*(unsigned int*)(h+page_offset+0x40)/sizeof(TYPE));
  close(map_file);
  return 0;
}
```

This example will be used multiple times, whether for qualifying the latencies of Xenomai or accessing the registers configured in the programmable logic (PL) of the Zynq.

Such direct memory accesses from userspace are possible – without writing a C program handling the `/dev/mem` interface and convert real to virtual addresses (`mmap`) – thanks to the `devmem` [10]. software. Hence for example, unlocking the Zynq configuration registers can be tested with

```
redpitaya> devmem 0xf8000004 32 0x767b   # password to lock
redpitaya> devmem 0xf800000c             # check status: 1=locked
0x00000001
redpitaya> devmem 0xf8000008 32 0xdf0d   # password to unlock
redpitaya> devmem 0xf800000c             # check status: 0=unlocked
0x00000000
```

5.1 Reconfiguring pin functions – handling hardware interrupts

The Red Pitaya only two provides two GPIO controled from the PS – the MIOs – to which LEDs are connected. These pins are hence not usable as inputs or for any other use than visual information. Other MIO pins are available, but their default configuration (`ps7_init.c` file generated by Vivado or provided with the Linux kernel for starting the system) is dedicated to alternative functions (SPI, UART).

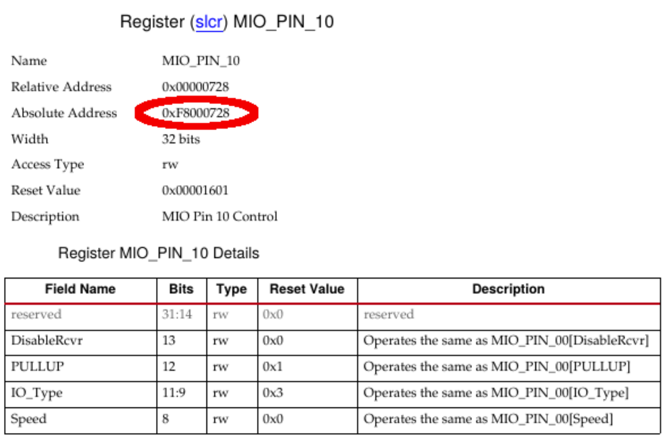
We wish to use some of these GPIOs to illustrate hardware interrupt handling. The GPIO functionality must hence be restored, for example when considering the SPI signal provided on MIO10, pin 3 of connector E2.

The Xilinx datasheet is clear on this topic: the function of a pin is defined with 3 bits named L0_SEL, L1_SEL and L2_SEL. We find as many configuration registers, and hence such bits, than pins to be configured. We identify the address associated with MIO10, check that its initial default configuration is indeed SPI communication, and restore it as a GPIO:

```
redpitaya> cd /sys/class/gpio/
redpitaya> echo "916" > export
redpitaya> echo "out" > direction
redpitaya> devmem 0xF800012C # GPIO clock active
```

[10]sources available in `buildroot` at `output/build/busybox-1.30.1/miscutils/devmem.c`

```
0x00500444
redpitaya> devmem 0xF8000728 # A = SPI
0x000016A0
redpitaya> devmem 0xF8000728 32 0x00001600
redpitaya> devmem 0xF8000728 # 0= GPIO
0x00001600
redpitaya> echo "1" > value
redpitaya> echo "0" > value
```

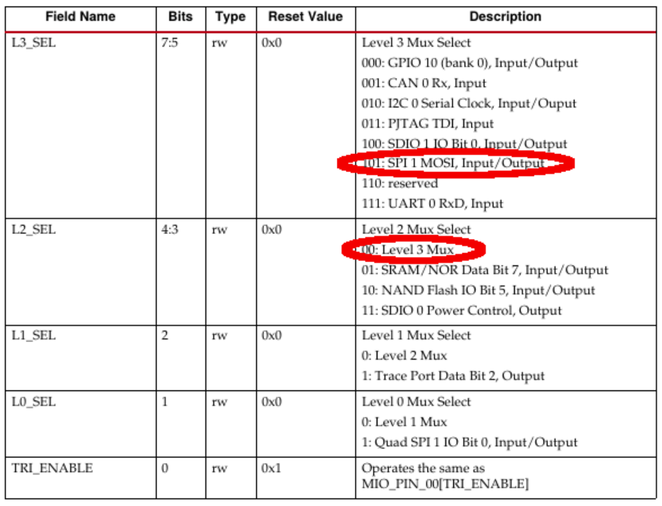


Register (slcr) MIO_PIN_10

Name	MIO_PIN_10
Relative Address	0x00000728
Absolute Address	0xF8000728
Width	32 bits
Access Type	rw
Reset Value	0x00001601
Description	MIO Pin 10 Control

Register MIO_PIN_10 Details

Field Name	Bits	Type	Reset Value	Description
reserved	31:14	rw	0x0	reserved
DisableRcvr	13	rw	0x0	Operates the same as MIO_PIN_00[DisableRcvr]
PULLUP	12	rw	0x1	Operates the same as MIO_PIN_00[PULLUP]
IO_Type	11:9	rw	0x3	Operates the same as MIO_PIN_00[IO_Type]
Speed	8	rw	0x0	Operates the same as MIO_PIN_00[Speed]



Field Name	Bits	Type	Reset Value	Description
L3_SEL	7:5	rw	0x0	Level 3 Mux Select 000: GPIO 10 (bank 0), Input/Output 001: CAN 0 Rx, Input 010: I2C 0 Serial Clock, Input/Ouput 011: PJTAG TDI, Input 100: SDIO 1 IO Bit 0, Input/Output 101: SPI 1 MOSI, Input/Output 110: reserved 111: UART 0 RxD, Input
L2_SEL	4:3	rw	0x0	Level 2 Mux Select 00: Level 3 Mux 01: SRAM/NOR Data Bit 7, Input/Output 10: NAND Flash IO Bit 5, Input/Output 11: SDIO 0 Power Control, Output
L1_SEL	2	rw	0x0	Level 1 Mux Select 0: Level 2 Mux 1: Trace Port Data Bit 2, Output
L0_SEL	1	rw	0x0	Level 0 Mux Select 0: Level 1 Mux 1: Quad SPI 1 IO Bit 0, Input/Output
TRI_ENABLE	0	rw	0x1	Operates the same as MIO_PIN_00[TRI_ENABLE]

Following these commands, the MIO10 pin (EP2 pin 3, Fig. 6) is indeed set as a GPIO, as demonstrated with the following Linux kernel module that uses the kernel *timer* to alternate the pin state.

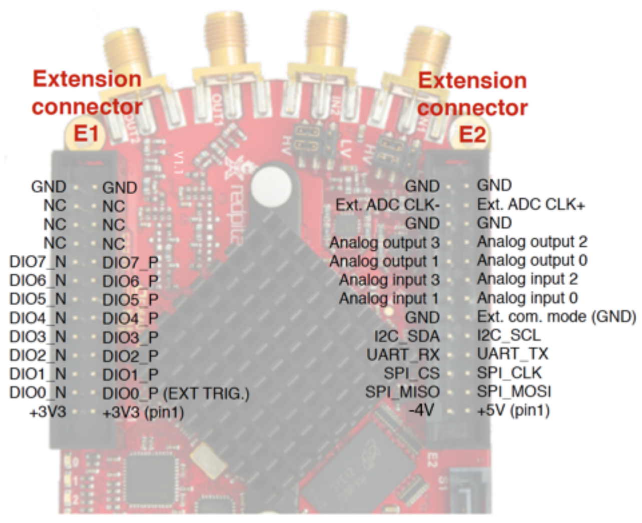


Figure 6: Redpitaya extention port pinout, from `https://redpitaya.readthedocs.io/en/latest/developerGuide/hardwa`
MIO10 is indicated here as SPI_MOSI or pin 3 of E2.

Notice that the argument provided to `gpio-lib` to configure the GPIO is the same than the one we provided earlier to `/sys/class/gpio`, namely 906+GPIO index (here 10) for pre-6 kernels and 512+GPIO index for 6.x kernels:

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/gpio.h>

struct timer_list exp_timer;

// int jmf_gpio=906+0; // 7 en conflit avec heartbeat => virer leds_gpio et led_class (sinon +0)
int jmf_gpio=906+10; // ou ('port'-'A')*32+pin dans devicetree
volatile int jmf_stat=0;

```

```c
static void do_something(struct timer_list *t)
{printk(KERN_INFO "plop");
 jmf_stat=1-jmf_stat;
 gpio_set_value(jmf_gpio,jmf_stat);
 mod_timer(&exp_timer, jiffies + HZ);
}

int hello_start(void);
void hello_end(void);

int hello_start()  // init_module(void)
{int delay = 1,err;

 printk(KERN_INFO "Hello\n");
 err=gpio_is_valid(jmf_gpio);
 printk(KERN_INFO "err: %d\n",err);
 err=gpio_request_one(jmf_gpio, GPIOF_OUT_INIT_LOW, "jmf_gpio"); // voir dans gpio.h
 printk(KERN_INFO "err: %d\n",err);

 // init_timer_on_stack(&exp_timer);
 // exp_timer.function = do_something;
 // exp_timer.data = 0;
 timer_setup(&exp_timer,do_something,0);
 exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks generated per →
    ↪second
 add_timer(&exp_timer);

 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 gpio_free(jmf_gpio);
 del_timer(&exp_timer);
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

Similary, this pin can trigger an asynchronous event (interrupt) its its voltage changes. An interrupt is handled in the sample kernel module as follows:

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio.h>

static int dummy, irq, jmf_gpio, dev_id;

void hello_end(void); // cleanup_module(void)
int hello_start(void); // cleanup_module(void)

static irqreturn_t irq_handler(int irq, void *dev_id)
{
        dummy++;
        printk(KERN_INFO "plip %d",dummy);
  return IRQ_HANDLED; // etait IRQ_NONE
}

int hello_start()  // init_module(void)
{int err;

 printk(KERN_INFO "Hello\n");
 jmf_gpio =906+10;  // PB2 : version devicetree ; 15 en version script.fex
 err=gpio_is_valid(jmf_gpio);
 err=gpio_request_one(jmf_gpio, GPIOF_IN, "jmf_irq");
 if (err!=-22)
    {printk(KERN_ALERT "gpio_request %d=%d\n",jmf_gpio,err);
     irq = gpio_to_irq(jmf_gpio);
     printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
     irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
    err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &dev_id);
    printk(KERN_ALERT "finished IRQ: error=%d\n",err);
    dummy=0;
   }
 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 free_irq(irq,&dev_id);
 gpio_free(jmf_gpio);  // libere la GPIO pour la prochaine fois
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

which is further optimized with interrupt handling at the kernel space aimed at sending a signal that might have registered with such a service:

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/fs.h>           // define fops
#include <linux/uaccess.h>
#include <linux/version.h>

#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/sched/signal.h>       // send_sig_info
#ifdef __ARMEL__
#include <linux/gpio.h>
//#include <linux/signal.h>           // do_send_sig_info
#else
#endif

int hello_start(void);
void hello_end(void);

int pid = 0;
#ifdef __ARMEL__
static int dummy,gpio,id;
static int irq;

static irqreturn_t irq_handler(int irq, void *dev_id)
#else
struct timer_list exp_timer;

static void irq_handler(struct timer_list *t)
#endif
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(5,0,0)
 struct siginfo sinfo; // siginfo
#else
 struct kernel_siginfo sinfo; // siginfo
#endif
 struct task_struct *task;
// alternative `a send_sig_info :
// struct pid *mypid;
// mypid= find_vpid(pid);
// if (mypid == NULL) pr_info("Cannot find PID from user program\r\n");
//    else kill_pid(mypid, SIGUSR1, 1);
#if LINUX_VERSION_CODE < KERNEL_VERSION(5,0,0)
 memset(&sinfo, 0, sizeof(struct siginfo));  // on cherche PID au cas ou` le process aurait →
    ↪disparu
#else
 memset(&sinfo, 0, sizeof(struct kernel_siginfo));  // on cherche PID au cas ou` le process aurait→
    ↪ disparu
#endif
 sinfo.si_signo = SIGUSR1;                    // depuis son enregistrement
 sinfo.si_code = SI_USER;
 task = pid_task(find_vpid(pid), PIDTYPE_PID);
 if (task == NULL) pr_info("Cannot find PID from user program\r\n");
    else send_sig_info(SIGUSR1, &sinfo, task);
#ifdef __ARMEL__
```

```c
 dummy++;
 printk(KERN_INFO "plip %d",dummy);
 return IRQ_HANDLED;
#else
 printk(KERN_INFO "plip %ld",jiffies);
 mod_timer(t, jiffies + HZ);
#endif
}

static int dev_open(struct inode *inod,struct file *fil);
static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
static int dev_rls(struct inode *inod,struct file *fil);

static struct file_operations fops=
{.read=dev_read,
 .open=dev_open,
 .write=dev_write,
 .release=dev_rls,};

int hello_start()  // init_module(void)
{int t=register_chrdev(91,"jmf",&fops);  // major = 91
#ifdef __ARMEL__
 int err;
#endif

 if (t<0) printk(KERN_ALERT "registration failed\n");
    else printk(KERN_ALERT "registration success\n");
 printk(KERN_INFO "Hello\n");

#ifdef __ARMEL__
 gpio =906+10;  // PB2 : version devicetree ; 15 en version script.fex
 err=gpio_is_valid(gpio);
 err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
 if (err!=-22)
    {printk(KERN_ALERT "gpio_request %d=%d\n",gpio,err);
     irq = gpio_to_irq(gpio);
     printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
     irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
     err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &id);
     printk(KERN_ALERT "finished IRQ: error=%d\n",err);
     dummy=0;
    }
#else
 timer_setup(&exp_timer,irq_handler,0); // was init_timer_on_stack(&exp_timer); -> replaced since 4.14
 exp_timer.expires = jiffies + HZ; // HZ specifies number of clock ticks generated per second
 add_timer(&exp_timer);
#endif
 return t;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
#ifdef __ARMEL__
 free_irq(irq,&id);
 gpio_free(gpio);  // libere la GPIO pour la prochaine fois
#else
 del_timer(&exp_timer);
#endif
 unregister_chrdev(91,"jmf");
}

static int dev_rls(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "bye\n");
 return 0;
}

static int dev_open(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "open\n");
 return 0;
}

static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
{char buf[15]="Hello read\n\0";
```

```
 int readPos=0;
 printk(KERN_ALERT "read\n");
 while (len && (buf[readPos]!=0))
   {put_user(buf[readPos],buff++);
    readPos++;
    len--;
   }
 return readPos;
}

static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
{int mylen;
 char buf[15];

 printk(KERN_ALERT "write ");
 if (len>14) mylen=14; else mylen=len;
 if (copy_from_user(buf, buff, mylen) == 0)
    sscanf(buf, "%d", &pid);

 printk(KERN_ALERT "PID registered: %d",pid);
 return len;
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");  // NECESSAIRE pour exporter les symboles du noyau linux !
```

in order to later share the information to userspace:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

void signal_handler(int signum)
{
    if (signum == SIGUSR1)
        printf("J'ai eu un signal\r\n");
    return;
}

int main()
{FILE *f;
 printf("Penser a mknod /dev/jmf c 91 0 avant de lancer ce programme\n");
    signal(SIGUSR1, signal_handler);
    printf("My PID is %d.\n", getpid());
    f=fopen("/dev/jmf","w");
    fprintf(f,"%d\n", getpid()); fflush(f);
    fclose(f);
    printf("send: %d\n", getpid());
    while (1) {};
    return 0;
}
```

6 Kernel programming: modules and communicating with userspace

Our objective in this section is to understand how to communicate between userspace and hardware by benefitting from kernel features. Although we have seen that various hardware abstractions (`script.bin` or `devicetree.dtb`) aim at hiding the hardware details to the developer, we will he consider the electronics engineer perspective who understands hardware and wishes to control from the Linux kernel to provide standardized interfaces to the user.

6.1 Why work at the kernel level?

Considering the abstraction layers between a userspace software and hardware, the kernel aims at keeping consistent the various requests to control the few hardware resources from the many possible applications. The kernel makes sure each task is given some time to execute (scheduler), that a unique task can access a hardware resource at any given time, and that messages read from the hardware are communicated to all userspace applications which requested access.

A module [3] is a piece of software that connects to the kernel to add features. Initially introduced with Linux kernel version 1.2 (March 1995), it avoids compiling the whole kernel when adding new features such as support for a new hardware peripheral. Most hardware peripherals could be accessible from userspace (unless Direct Memory Access or interrupts are involved) but doing so would waste the consistency of a single software driving a peripheral provided by the kernel supervision (for example by preventing multiple copies of a module accessing the same hardware being loaded). Some features, such as the mentioned Direct Memory Access or interrupts, are just not accessible outside the kernel.

When configuring the kernel, selecting a module is achieved by setting to `M` such options when running `make menuconfig`. Compiling modules is completed with `make modules` followed by `make modules_install` which copies the copilation result to `/var/lib/modules`.

These options are accessed when using buildroot by executing `make linux-menuconfig` so that Buildroot's `make` will compile both kernel and modules, and store the result in the image to be stored on the SD card.

Any driver developer must understand the architecture of a kernel module in order to provide the interfaces expected by a user willing to access hardware without understanding the details of hardware configuration.

The two classes of kernel modules are designed for transfering data blocks (for example to a mass storage interface such as a hard disk), or as individual bytes [11] We will focus on the latter, easier to grasp and meeting most sensor applications while handling hardware peripherals easier to run during demonstrations.

All communication is achieved through pseudo-files located in the `/dev/` directory. A file can be opened, closed, and data can be written to or read from the file. An additional method which does not have any meaning in terms of file access is the input output configuration: `ioctl` or (Input/Output Control).

6.2 Structure of a kernel module and compilation

The minimum needed for linking a module to a Linux kernel are the functions for initializing resources and a function for freeing these resources. The following examples provides a skeleton and become familiar with the Linux kernel tree structure on the one hand, and compilation method through a `Makefile` requesting methods associated with the kernel being run on the target platform.

A minimalisting kernel module is provided below, for becoming familiar with the initialization function and compilation mechanism as well as linking the module with the kernel.

```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

static int __init hello_start(void){printk(KERN_INFO "Hello\n");return 0;}
static void __exit hello_end(void) {printk(KERN_INFO "Goodbye\n");}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");
```

Listing 1: First example of kernel module

The kernel module defines the entry and exit functions. When a module is linked to the kernel (`insmod my_mod.ko`), the `init()` method is called. When the module is unlinked from the kernel (`rmmod my_mod`), the `exit` mothod is called. Both methods are macros calling the `init_module()` and `cleanup_module()` functions as defined at voir `linux/init.h` in the Linux kernel source code.

While historically the `.o` object was directly linked to the kernel, since its 2.6 version the Linux kernel requires additional informations to link with a *kernel object* which differs between the `.o` and `.ko`. Although the Makefile declares a dependency on the `.o` output, the `.ko` file is indeed the one to be loaded using `insmod` as described at `https://tldp.org/HOWTO/Module-HOWTO/linuxversions.html`: "In Linux 2.6, the kernel does the linking. A user space program passes the contents of the ELF object file directly to the kernel. For this to work, the ELF object image must contain additional information. To identify this particular kind of ELF object file, we name the file with suffix ".ko" ("kernel object") instead of ".o" For example, the serial device driver that in Linux 2.4 lived in the file serial.o in Linux 2.6 lives in the file serial.ko."

[11] `https://appusajeev.wordpress.com/2011/06/18/writing-a-linux-character-device-driver/`

A module is necessarily linked with a given and known version of the kernel, and will have to be recompiled whenever the latter is updated. Buildroot provides a consistent development framework including the cross-compilation toolchain and the kernel sources, which are found at `output/build/linux-`*version* of buildroot. In our case, the Linux kernel is fetched using `git` and the directory name is postfixed with the hash key.

As a first step, we make sure the compiler is in the executable path:

```
export PATH=$PATH:$HOME/.../buildroot/output/host/usr/bin/
```

then we compile the module – assumed to be named `mymod.c` [12] to become `mymod.ko` – using the following `Makefile`:

```
obj-m += mymod.o

all:
  make ARCH=arm CROSS_COMPILE=arm-linux- -C \
  /home/jmfriedt/buildroot/output/build/linux-[...] M=$(PWD) modules

clean:
  rm mymod.o mymod.ko
```

This `Makefile` is analyzed as follows:

1. it calls itself another Makefile since executing the `make` command
2. this other Makefile is located in the sources of the kernel we will be linking and is selected thank to the `-C directory` option which indicates the location of the targeted configuration,
3. we request the `modules` method since this command could be summarized as `make modules` with a few additional options...
4. ... amongst which the cross-compilation towards and ARM target, hence the `ARCH` option and the compiler prefix in `CROSS_COMPILE`.
5. Finally, compiling a module requires providing the location in which the sources are located: here we are considering the working directory so that `M=$(PWD)` in this case.
6. If the same module is to be compiled on the PC (host architecture), we remove `ARCH` and `CROSS_COMPILE` to only keep `make -C kernel_directory M=$(PWD) modules` (see below)

A kernel module is no only compiled to the embedded target board, but also possibly to the host as long as low-lever hardware is not accessed. In order to know the kernel version running on the PC, run the `uname -a` command. Make sure the kernel sources are available, or at least the associated headers and configuration, for example with the package `linux-headers-amd64` when using Debian GNU/Linux. On the PC, the Makefile will look like

```
obj-m += t.o
all:
      make -C /usr/src/linux-headers-6.10.4-amd64 M=$(PWD) modules
```

making sure **not** to use the `common` extension which would not provide the configuration of this particular kernel for the current host architecture.

We observe how powerful Buildroot is in providing a consistent working environment with the cross-compilation toolchain (`CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf-`) and the kernel tree structure (`buildroot-2024.05.2.tar.gz/output/build/linux-xilinx-v2024.1`) which is used when compiling the module. The principle of the `Makefile` is to add the list of kernel modules under development (`mymod`) to the list of kernel modules, and run the compilation of the kernel with `make`. The other kernel objects having been already compiled, only our current module will be generated.

⚠ **Warning**: when using a board running a kernel other than the one provided by Buildroot, synchronizing the kernel version with the one running on the board is mandatory. Doing so is achieved by fetching on the embedded board the Linux configuration found in `/proc/configs.gz`, copy this file in the `.config` configuration file of the kernel being compiled by Buildroot, run `make linux-menuconfig` to tell Buildroot of the update, and finally `make` from the top Buildroot location to recompile the full kernel and dependencies. Once the compilaiton is completed, the module we will compile from now on will be compatible with the kernel running on the board.

[12] **do not call the module** `kernel.c`: doing so will lead to a successful compilation, but by using an object that does not result from our own source code but from a file located in the kernel tree structure!

The module is linked to the kernel with `insmod mymod.ko` and removed with `rmmod mymod`. The list of modules linked to the kernel is displayed with `lsmod`. The messages from the kernel, including its modules, are displayed on the console (in the case of the Red Pitaya, on the serial port) or in the kernel logs displayed with `dmesg` or `cat /var/log/messages` (or `/var/log/syslog`).

The result of the kernel module compilation is transfered to the Red Pitaya board (NFS or `scp`) to be linked to the kernel with `insmod mymod.ko`. We validate that the command was acknowledged by displaying the log messages with `dmesg`:

```
# insmod mymod.ko
# dmesg | tail -1
[  249.879087] Hello
# rmmod mymod.ko
# dmesg | tail -1
[  256.571319] Goodbye
```

These information are also found in `/var/log/messages` which can be constantly monitored using `tail -f`.

When a module compilation fails, it is fundamental to remember that a kernel module is linked against a given kernel configuration. If this configuration is lost, is can always be recovered with `scripts/extract-ikconfig` whose output should be saved in a `.config` file allowing to regenerate a working environment consistent with the one expected by the kernel being executed.

The communication capabilities of our module are however limited: we shall extend such capabilities with a character device interface (*char device*).

6.3 Communication through `/dev`: interacting with the user

A module only able to load and unload is hardly usable. In order to interact with the user, we must provide methods for reading or writing. The standard location of the pseudo-files allowing such interaction between the kernel and userpace is in the `/dev` directory. These files act as pipes between the two spaces (kernel and user) through which single bytes (characters) are shared (as opposed to data blocks). The administrator (root or `sudo` on the host PC) can create such pseudo-files using the command `mknod /dev/jmf c 90 0` with the arguments indicating that we are creating a *character device* (`c`) identified with the major number 90 (kind of peripheral) and minor number 0 (index in the list of such peripherals). On the PC, adding `-m 666` to the node creation will allow for read/write permissions to all users: `mknod /dev/jmf c 90 0 -m 666`.

⚠ In case a resource busy message is displayed when inserting the module (error -16 – EBUSY), the major number 90 might alread be used by another module: even if it does not appear in the list of entries in `/dev`, the list of peripherals in `/proc/devices` will provide all the major numbers already allocated to kernel modules. For example on the Red Pitaya, major number 90 is allocated to the `mtd` module. On a host PC, the 99 major number is allocated to `ppdev`.

The structure defining which function is called when a system call is transfered from userspace to kernel space (function pointer) when opening or closing the pipe (`open` and `close` in userspace) as well as reading and writing data through the pipe (`write` and `read` system calls from userspace respectively) is name `file_operations`.

In the example below, writing to `/dev/jmf` will lead to a message in the kernel logs including the string that was written by the user:

```c
// mknod /dev/jmf c 90 0
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/fs.h>           // define fops
#include <linux/uaccess.h>

static int dev_open(struct inode *inod,struct file *fil);
static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
static int dev_rls(struct inode *inod,struct file *fil);

char buf[15]="Hello read\n\0";

```

```c
int hello_start(void); // declaration pour eviter les warnings
void hello_end(void);

static struct file_operations fops=
{.read=dev_read,
 .open=dev_open,
 .write=dev_write,
 .release=dev_rls,
};

int hello_start()  // init_module(void)
{int t=register_chrdev(90,"jmf",&fops);  // major = 90
 if (t<0) printk(KERN_ALERT "registration failed\n");
    else printk(KERN_ALERT "registration success\n");
 printk(KERN_INFO "Hello\n");
 return t;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 unregister_chrdev(90,"jmf");
}

static int dev_rls(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "bye\n");return 0;}

static int dev_open(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "open\n");return 0;}

static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
{int readPos=0;
 printk(KERN_ALERT "read\n");
 while (len && (buf[readPos]!=0))
   {put_user(buf[readPos],buff++);
    readPos++;
    len--;
   }
 return readPos;
}

static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
{int m=0,mylen;
 printk(KERN_ALERT "write ");
 if (len>14) mylen=14; else mylen=len;
 for (m=0;m<mylen;m++) get_user(buf[m],buff+m);
 // DO NOT try with vvv: kernel panic since accessing usespace from kernel space
 // for (m=0;m<mylen;m++) buf[m]=buff[m];
 buf[mylen]=0;
 printk(KERN_ALERT "%s \n",buf);
 return len;
}

module_init(hello_start);
module_exit(hello_end);
```

This module is loaded and its functions called using

```
# echo "Hello" > /dev/jmf
[  258.408111] open
[  258.414953] write
[  258.416874] Hello
[  258.416874]
[  258.420911] bye
# dd if=/dev/jmf bs=10 count=1
[  282.774521] open
[  282.776557] read
Hello[  282.778454] bye

0+1 records in
0+1 records out
```

```
# echo "toto" > /dev/jmf [  290.922228] open

[  290.924908] write
[  290.926933] toto
[  290.926933]
[  290.930904] bye
# dd if=/dev/jmf bs=10 count=1
[  293.484720] open
[  293.486761] read
toto[  293.488659] bye

0+1 records in
0+1 records out
```

This example includes a modification to the module shown previously so that the displayed message is the one read previously: implement this modification.

This module implements communication between user space and kernel space through the pseudo-file named `/dev/jmf`. A file located in the `/dev` is not a file for storing data but a link allowing for sharing data and system calls between user space and kernel space. This file was created with `mknod /dev/jmf c 90 0` with 90 its major number identified and 0 its minor number identifier. The major number remains constant for all files related to a given type of peripheral, while the minor number indexes which peripheral is considered by being incremented for each new instance. We check that the association of the communication peripheral with the module is correct by reading `/proc/devices` which includes the `jmf` entry associated with the major number we selected. (90, which is free as verified with `ls -l /dev/`). However, the `/dev/` is ofter dynamically generated during the boot sequence, and any new entry will have to be recreated after a reboot sequence.

An alternate solution to explicitly providing the communication entry in `/dev` by manually selecting the major number is to use the *misc device* class. Doing so is achieved by creating the global variable

```
 struct miscdevice jmfdev;
```

and in the init function defining:

```
 jmfdev.name = "jmf";  // /dev/jmf using the major number of the misc class
 jmfdev.minor = MISC_DYNAMIC_MINOR;
 jmfdev.fops = &fops;
 jmfdev.mode = 0666; // permissions for the user to access
 misc_register(&jmfdev);  // dynamic creation of the /dev/jmf entry
```

which is concluded in the exit function with

```
 misc_deregister(&jmfdev);
```

6.4 The `ioctl` system call

After defining the systems calls matching the requirement that all peripherals must appear as files, it occured that one more system call had to be added that breaks this initial unix design philosophy: this new system call is called `ioctl` for input/output control. This system call has been added to complement functionalities that cannot be solved with just `read` and `write`, namely configuration of the peripherals (for example, tuning the sampling rate or the data size of a sound card).

The previous example listing can hence be complemented with the following functions:

```
// static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
//      unsigned long arg); // pre-2.6.35
static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg); // actuel

static struct file_operations fops=
{.read=dev_read,
 .open=dev_open,
 .unlocked_ioctl=dev_ioctl,
 .write=dev_write,
 .release=dev_rls,};

// static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
//    unsigned long arg) // pre-2.6.35
static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{switch ( cmd )
 {case 0: printk(KERN_ALERT "ioctl0");break;
  case 1: printk(KERN_ALERT "ioctl1");break;
  default:printk(KERN_ALERT "unknown ioctl");break;
 }
 return 0;
}
```

Executing the following program – we are not aware of any way of calling `ioctl()` from the shell without writing a dedicated C program – which we call `ioct`

```
#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */
#include <stdio.h>
#include <stdlib.h>

#define IOCTL_SET_MSG 0
#define IOCTL_GET_MSG 1
#define DEVICE_FILE_NAME "/dev/jmf"

ioctl_set_msg(int file_desc, char *message)
{ int ret_val;
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
  printf("set_msg message: %s\n", message);
}

ioctl_get_msg(int file_desc,char *message)
{ int ret_val;
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
  printf("get_msg message: %s\n", message);
}

main(int argc,char **argv)
{ int file_desc, ret_val;
  char msg[30] = "Message passed by ioctl\n";

  file_desc = open("/dev/jmf", 0);
  printf("%d\n",file_desc);
  if (argc>1) sprintf(msg,argv[1]);
  msg[4]=0;
  ioctl_set_msg(file_desc,msg);

  sprintf(msg,"Hello World");
  ioctl_get_msg(file_desc,msg);
  close(file_desc);
}
```

will generate the following sequence:

```
# lsmod
Module                  Size  Used by
g_ether                44895  0
# insmod mymod.ko
# ./ioctl
# tail /var/log/messages
Jan  1 00:11:45 buildroot kern.alert kernel: [  705.740703] registration success
Jan  1 00:11:45 buildroot kern.info kernel: [  705.745730] Hello
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.805798] open
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.808491] ioctl1
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.818059] ioctl0
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.820838] bye
```

From a user perspective, the `ioctl()` function is called exactly as `read()` or `write()` would be, but will handle 3 arguments: the file descriptor, the ioctl index (consistent with the declaration of the kernel module, here 0 or 1) and possibly an argument. The second page of the `ioctl` manual is clear about these aspects. From the kernel side, transfering arguments [13] towards userspace memory addresses requires using `put_user()` for a single scalar value or `copy_to_user` for an array (memory area from kernel space to user space).

6.5 From virtual addresses to physical addresses

Although many embedded systems do not bother with hardware handling of memory through a Memory Management Unit – MMU – a multitasking operating system such as Linux benefits significantly from the use of such a peripheral and justifies the selection of a microcontroller powerful enough to be fitted with an MMU. Memory organization, handled by the MMU, should not be cared of by the developer as long as memory allocation and access remain consistent, *except* when accessing a known physical memory location, as for exemple required when reaching hardware configuration registers. In that cas, we must perform the reverse conversion, from the virtual memory representation to the real physical addressing. From a userspace perspective, such a functionality is provided by `mmap()`. From the kernel space, it is `ioremap()`.

Electronics engineers are only aware of physical addresses for reaching a peripheral: this is the address associated with each register in the microcontroller datasheet. The computer scientist only knows virtual addresses, as handled by the operating system running on top of the MMU. We must provide a link between these two worlds in order to let the communicate.

At the kernel leve, the instruction [3, chap.9] in charge of this link is `void *ioremap(unsigned long phys_addr, unsigned long size);`

[13] `http://www.makelinux.net/ldd3/chp-6-sect-1`

We will illustrate its use by complementing the methods called when inserting a module to switch on a LED, and switch off the LED when the module is removed from memory. This example is for demonstration purposes only since in a practical application, the module `linux-*/drivers/gpio/gpio-xilinx.c` implements such functionalities, hiding the subtelties of hardware access from the developer.

The resources needed to understand the organization of the registers needed to configure GPIOs of the 14-bit Red Pitaya are described at `http://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html` with the help of the unavoidable – but rather lengthy – `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. The addresses of the registers related to GPIO configuration are described in the chapter 14 of the user manual.

We complement the kernel module written so far with

```
static void __iomem *jmf_gpio;   //int jmf_gpio;
#define IO_BASE2 0xe000a000

int hello_start()  // init_module(void)
{int delay = 1;
 unsigned int stat;
[... gestion horloge GPIO ...]

 if (request_mem_region(IO_BASE2,0x2E4,"GPIO test")==NULL)
     printk(KERN_ALERT "mem request failed");
 jmf_gpio = (void __iomem*)ioremap(IO_BASE2, 0x2E4);
 writel(1<<mio,jmf_gpio+0x204);
 writel(1<<mio,jmf_gpio+0x208);
 writel(1<<mio,jmf_gpio+0x40);

 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 release_mem_region(IO_BASE2, 0x2e4);
}
```

after remembering to include the header files `linux/io.h` and `linux/ioport.h`. The code can only be understood by reading [4, p.1347] partly reproduced here, with the description of the various registers handling GPIO behaviour, and [4, p.1580] for the clock handling part.

6.6 Adding a periodic timer

The Linux kernel provides access to timers for clocking tasks. We use such resources to make the LED periodically blink and display a message in the kernel log files.

In order to inform the user that the Red Pitaya board is alive, we wish to display a heartbeat pattern on the LED blinking periodically. The Linux kernel provides the timer as described at [3, chap.7] and illustrated in the following code:

```
struct timer_list exp_timer;

static void do_something(struct timer_list *t)
{printk(KERN_ALERT "plop");
 mod_timer(&exp_timer, jiffies + HZ);
}

int hello_start()  // init_module(void)
{...
 // init_timer_on_stack(&exp_timer);
 // exp_timer.function = do_something;
 // exp_timer.data = 0;
 timer_setup(&exp_timer,do_something,0);
 exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks/second
 add_timer(&exp_timer);
...
}

void hello_end() // cleanup_module(void)
{...
 del_timer(&exp_timer);
}
```

with `jiffies` the internal representation in the kernel of the current time.

We observe in `dmesg | tail` that the timer is indeed periodically triggered, once every second, as requested by `mod_timer(&exp_timer, jiffies + HZ);`:

```
[ 5697.089643] registration success
[ 5697.093780] Hello
[ 5698.094083] plop
[ 5699.094146] plop
[ 5700.094227] plop
[ 5701.094286] plop
[ 5702.094348] plop
```

6.7 Semaphore, mutex and spinlock [5]

A driver providing a `read` method continually communicates with the userspace process requesting the information (`cat < /dev/mydriver`). Such a condition is not representative of a practical application, in which the producer requires some time to generate data that are the be sent to userspace: the `read` method hence behaves as a data consumer. In this producer-consumer architecture, some mechanism for blocking on the one hand the call to `read` as long as the data have not been produced, and on the other hand to make sure the access to the memory area shared by the producer and consumer remains consistent, is needed. The first functionality is provided by the semaphore, and the second by the mutex and its faster implementation (but using more resources) the spinlock.

6.7.1 Semaphore

The semaphore acts here as a counter remembering how many times data were produced. A consumer trying to fetch data decrements the semaphore: if the semaphore is already 0, then the consumer (here the `read` method) is blocked until the producer has incremented the semaphore. A semaphore incremented multiple times allows the consumer to obtain multiple datasets.

The definition of the constants and prototypes used by semaphores are summarized with

```c
#include <linux/semaphore.h>
struct semaphore mysem;
sema_init(&mysem, 0);        // init the semaphore as empty
down (&mysem);
up (&mysem);
```

Simulate the priodic production of datasets leading to the increment of the semaphore in a timer handler. Block the call to `read` as long as datasets have not been procduced.

What happens when reading from a peripheral accessible in the `/dev/` entry when the semaphore has already incremented multiple times?

Amongst the five core functions of a communicating module – init, exit, open, read, release – which one allows producing datasets only when needed by the user? Demonstrate how the previous program is possibly modified to produce datasets (incrementing the semaphore) only when requested by the user, and not as early as loading the module.

6.7.2 Mutex and spinlock

Data produced are stored in a buffer memory for sharing information between producer and consumer. A mechanism must handle the consistency between writing by the producer and reading by the consumer: the buffer memory must not be overwritten while being read, and its content should not be read as it is being written. The MutEx (*MUTually EXclusive*) provides a binary lock – locked or unlocked – preventing two processes from simultaneously accessing the same memory area or variable. Before reaching a buffer memory, each process (producer or consumer) locks the mutex, preventing the other process which might wish to lock the mutex itself from continuing execution. Once the operation – reading or writing – completex, the mutex is unlocked and allows resuming of the waiting process.

The mutex mechanism allows for putting a task in a sleep state until the mutex is unlocked. Making the task sleep frees some processor resources, but requires changing context which might be lengthy and requiring significant processing resources if the read and write operations should be perform with short latencies. An alternative solution is the spinlock, which operates similarly but keeps the task in an active state by continuously probing the state of the lock: in this case, the processor is continuously active, mais le latency is lowered and no context change is needed.

The definitions of constants and prototypes of mutexes are as summarized with

```c
#include <linux/mutex.h>
struct mutex mymutex;
mutex_init(&mymutex);
mutex_lock(&mymutex);
mutex_unlock(&mymutex);
```

The definitions of constants and prototypes of spinlocks are as summarized with

```c
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(myspin);
spin_lock_init(&myspin);
spin_lock(&myspin);
spin_unlock(&myspin);
```

Demonstrate an implementation of a mutex designed to provide consistency between reading and writing a common variable. The text displayed by a call to the `read` function should be the one stored by a call to `write` with the addition of the value of a counter periodically incremented to simulate the production of data.

Instead of updating the content of an array of characters to be displayed in the timer handler itself, create a task which is scheduled when time allows and is triggered by the timer: place the processing steps requiring most computational power (mutex locking and unlocking, updating the buffer content) in this task.

6.8 sysfs

The interaction between a module and a user is from now on no longer handled by a pseudo-file (char or block device) in the `/dev` directory but by a `/sysfs` in example 6.8: a directory in `/sys` is dynamically created when loading the module. Hence, a `platform_device` initializes a communication entry in `/sys`, here called `gpio-simple`. The `probe` method is called upon initialization of the platform device with the same driver identifier defined in the ".name" structure: the first argument of `platform_device_register_simple` must be the same as the driver identifier:

```c
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/platform_device.h>
#include <linux/err.h>

static struct platform_device *pdev;

static int gpio_simple_probe(struct platform_device *pdev)
{
  return 0;
}

static int gpio_simple_remove(struct platform_device *pdev)
{
  return 0;
}

static struct platform_driver gpio_simple_driver = {
        .probe          = gpio_simple_probe,
        .remove         = gpio_simple_remove,
        .driver = {
                .name   = "gpio-simple",
        },
};

static int __init gpio_simple_init(void)
{ int ret;
  ret = platform_driver_register(&gpio_simple_driver);
  if (ret) return ret;
  pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
  if (IS_ERR(pdev))
    {platform_driver_unregister(&gpio_simple_driver);
     return PTR_ERR(pdev);
    }
 return 0;
}

static void __exit gpio_simple_exit(void)
{
  platform_device_unregister(pdev);
  platform_driver_unregister(&gpio_simple_driver);
}

module_init(gpio_simple_init)
module_exit(gpio_simple_exit)

MODULE_DESCRIPTION("GPIO simple driver");
MODULE_LICENSE("GPL");
```

Add the display of messages in the `probe`, `init`, `exit` and `remove` methods, and observe in which order they are called, most significantly the `probe` function when the platform_device is created and calls the associated driver.

We then add pseudo-files for communicating with the user, accessible at
`/sys/bus/platform/drivers/gpio-simple`

This result is achieved by adding a structure holding the pointer to the communication functions and initialized with the `DEVICE_ATTR` macro.

```
static ssize_t
gpio_simple_show(struct device *dev, struct device_attribute *attr, char *buf)
{
        return sprintf(buf, "%d\n", gpio_get_value(gpio));
}

static DEVICE_ATTR(value, 0444, gpio_simple_show, NULL);

static int gpio_simple_probe(struct platform_device *pdev)
{
  err = device_create_file(&pdev->dev, &dev_attr_value);
  if (err < 0)
    goto err_free_irq;

  return 0;
  [...]
}

static int gpio_simple_remove(struct platform_device *pdev)
{
  device_remove_file(&pdev->dev, &dev_attr_value);
  [...]
}
```

The code becomes more challenging to read unless analyzing the source code of
`.../buildroot/output/build/linux-headers-XXX/include/linux/device.h` and observing that the macro is indeed defined as

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
        struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

Hence, this macro creates in our case a structure named `dev_attr_value` (`value` being the name provided as first argument to the macro) provided as argument to the `device_create_file()` function.

Finally, we add the timer handler and trigger the associated event with:

```
static struct work_struct work;

static void gpio_simple_notify(struct work_struct *ws)
{pr_info("IRQ %d triggered on GPIO %d, value=%d\n",
}

static void do_something(struct timer_list *t)
{printk(KERN_INFO "plop: %lu",jiffies);
 schedule_work(&work);
}

static int gpio_simple_probe(struct platform_device *pdev)
{[...]
 INIT_WORK(&work, gpio_simple_notify);
}

static int gpio_simple_remove(struct platform_device *pdev)
{device_remove_file(&pdev->dev, &dev_attr_value);
 cancel_work_sync(&work);
  [...]
}
```

as well as the communication with the entry point `value` in the associated `/sys` filesystem. In this example, a task is triggered when an interrupt is triggered. This event will unlock the read function called by a userspace process when an interrupt will be triggered.

Demonstrate how a file being read in the `sysfs` is unblocked every time the timer reaches its threshold and launches a tasklet in charge of unlocking a semaphore.

7 Compiling a Linux image Linux with Xenomai support (real-time extension but no DTBO support)

Xenomai has been ported to the Zynq and most significantly J.H. Brown & B. Martin, *How fast is fast enough? Choosing between Xenomai and Linux for real-time applications*, Proc. 12th Real-Time Linux Workshop (RTLWS'12) (2012). The updates brought to an official mainline kernel were adapted to the `linux-xlnx` kernel used here and included as patches to be applied when compiling Xenomai from Buildroot.

Configuring buildroot with Xenomai support for the Red Pitaya is called with the `redpitaya_xenomai_defconfig` configuration file. Hence, in the buildroot directory and after exporting the `BR2_EXTERNAL` variable towards the `redpitaya` support files, run the `make redpitaya_xenomai_defconfig` command to load the configuration, which is then applied during the compilation with `make`. As before, after quite some time and additional storage space use, the resulting image is located in `output/images` and is transfered to the SD card with the appropriate `dd` command.

In case the FPGA (PL) of the Zynq is to be used, make sure to activate its support in the kernel with `make linux-menuconfig` and after searching for FPGA, select *FPGA Configuration Framework* and in the sub-menu:

```
< >   FPGA debug fs
< >   Altera Partial Reconfiguration IP Core
< >   Altera FPGA Passive Serial over SPI
< >   Altera CvP FPGA Manager
<*>   Xilinx Zynq FPGA
< >   Xilinx Configuration over Slave Serial (SPI)
< >   Lattice iCE40 SPI
< >   Lattice MachXO2 SPI
<*>   FPGA Bridge Framework
< >     Altera FPGA Freeze Bridge
< >     Altera FPGA Freeze Bridge
<*>   Xilinx LogiCORE PR Decoupler
<*>   FPGA Region
<*>     FPGA Region Device Tree Overlay Support
< >   FPGA Device Feature List (DFL) support
```

as well as activating

```
Device Tree and Open Firmware support
-*-   Device Tree overlays
```

Following the compilation and execution of this new image, new messages are displayed upon booting Linux confirming that Xenomai and associated tools have been included in the kernel:

```
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 4398046511103ns
I-pipe, 333.333 MHz clocksource, wrap in 12884 ms
```

and

```
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
[Xenomai] scheduling class idle registered.
[Xenomai] scheduling class rt registered.
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt v3.0.5 (Sisyphus's Boulder)
workingset: timestamp_bits=30 max_order=17 bucket_order=0
```

A classical tool is provided by Xenomai called `latency` for qualifying delays of the real-time operating system, in which the generic Linux kernel has only become one additional non-real time task:

```
redpitaya> latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -2.798|     -2.435|      2.714|       0|     0|     -2.798|      2.714
RTD|     -2.811|     -2.374|      7.931|       0|     0|     -2.811|      7.931
RTD|     -2.380|     -1.888|      6.692|       0|     0|     -2.811|      7.931
RTD|     -2.806|     -2.393|      6.939|       0|     0|     -2.811|      7.931
RTD|     -2.836|     -2.397|      5.680|       0|     0|     -2.836|      7.931
RTD|     -2.870|     -2.358|      7.046|       0|     0|     -2.870|      7.931
RTD|     -2.883|     -2.388|      8.137|       0|     0|     -2.883|      8.137
```

Negative values – meaningless – are documented at `https://embeddedgreg.com/2017/07/16/getting-started-with-xeno` `comment-page-1/` and we indeed find on the Red Pitaya the pseudo-filesystem `/proc/xenomai`, as well as the `autotune` tool in `/tmp/xeno_zynq/usr/xenomai/sbin` used for tuning the system parameters [14], in order to finally achieve a meaningful result:

```
redpitaya> latency
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      0.993|     10.388|     26.247|       0|     0|      0.993|     26.247
RTD|      0.980|      8.411|     29.984|       0|     0|      0.980|     29.984
RTD|      1.137|      9.887|     25.449|       0|     0|      0.980|     29.984
RTD|      1.180|     11.129|     26.790|       0|     0|      0.980|     29.984
RTD|      1.152|     10.576|     28.543|       0|     0|      0.980|     29.984
RTD|      1.055|     10.865|     27.385|       0|     0|      0.980|     29.984
RTD|      1.063|     10.913|     28.225|       0|     0|      0.980|     29.984
```

Another tool called `stress` allows to load the system and observe that latencies remain stable despite the resources requested by this program.

- check on your system that the real time options have been activated, and most significantly the associated tools (*testsuite* and `rt-tests`). Achieving this result requires activating `Target packages` → `Real-Time` → `Xenomai Userspace` → `Install testsuite`. Furthermire, manually select the Xenomai version by filling the `Custom Xenomai version` field with `3.0.5`,
- some additional shell tools might be wisely added, for example `screen` in `Target packages` → `Shell and utilities` → `screen` or `Target packages` → `System tools` → `cpuload`.

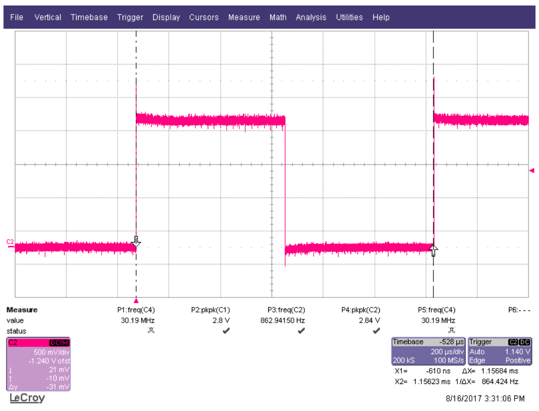
The latencies of the real time system [6] are qualified with `echo "0" > /proc/xenomai/latency` followed with `latency -p 100`. Observe the latency fluctuations when, in another terminal, the processor is loaded with `while ( true ) ; do echo toto;done`. This last command is run in a second shell started from `screen` (CTRL-A c to create a new shell session, CTRL-A a to switch between sessions).

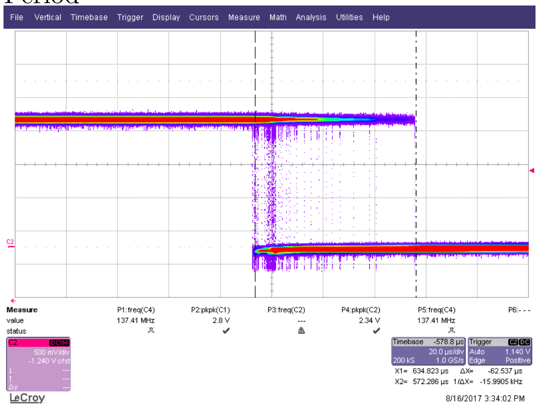
[14] `autotune --period 10000`

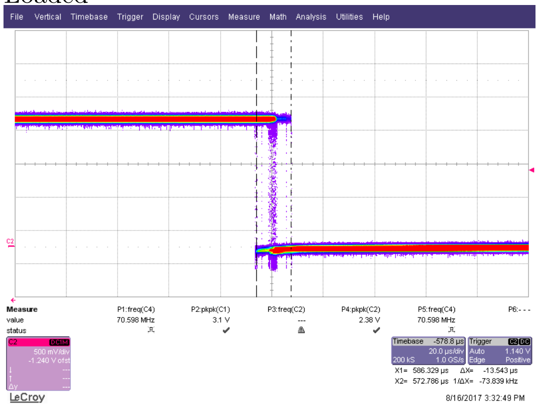
8 Latency qualification

The four examples explicited below investigate how different delays between toggling LED state either by calling the `sleep` function (active delay), or by calling a timer callback function. Without Xenomai, calling signal callbacks leads to the worst stability as soon as the processor becomes loaded. In all other cases, quantifying latency is achieved by triggering an oscilloscope on the rising edge of the LED state and watching the falling edge delay. If the delay met exactly the expected duration, all falling edges would overlap. The time interval over which the falling edges are observed quantifies the resistance of each method to processing load and the ability to meet the expected latency requirements under the various delay conditions. The case of a kernel module is not addressed.

8.1 Delay using `sleep`, without Xenomai


Period


Loaded

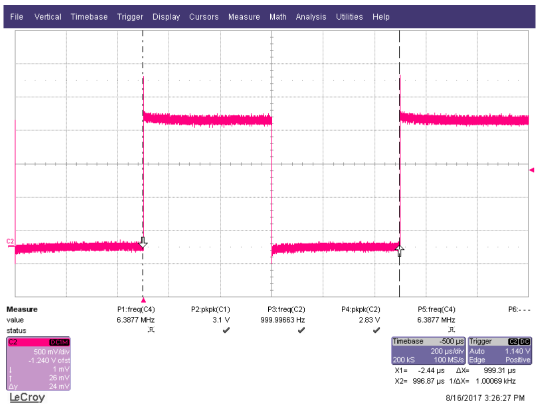

Unloaded

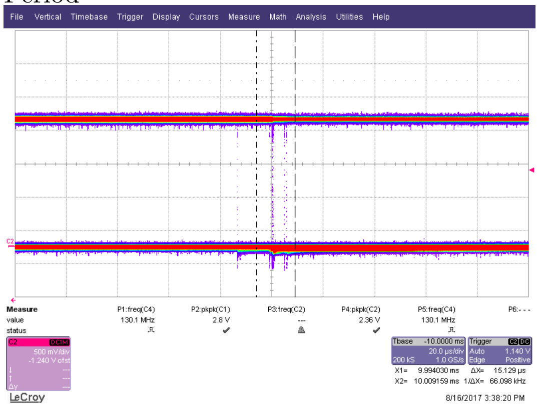
```c
#include <stdlib.h>
#include <stdio.h>
#include "ledlib.h"

int main(int argc, char **argv)
{int status=0,GPIO;
 unsigned char *h;
 if (argc>1)
    GPIO=atoi(argv[1]);
 else
    GPIO=10;
 h=red_gpio_init(GPIO);
 red_gpio_set_cfgpin(h,GPIO); // LED0

 while (1) {
    red_gpio_output(h,GPIO,status); usleep(TIMESLEEP);
//    printf("status=%x\n",status);
    status^=0xff;
 }
 red_gpio_cleanup();
 return 0;
}
```

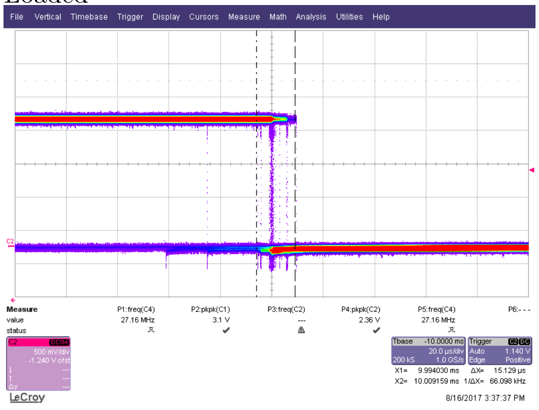
This program follows the previous example seen earlier of toggling LED state, with a 500 ms delay. In this and all the following examples, the processor is loaded with the following command executed in a second terminal on the Red Pitaya: `stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s` executed 5 times in a raw for a total measurement duration of about a minute.

8.2 Delay using a timer, without Xenomai

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "ledlib.h"

volatile int status=0,GPIO; // declenche' a chaque appel du timer => exterieur
volatile unsigned char *h;

void test(int signum) {
    red_gpio_output((unsigned char*)h,GPIO,status);
    status^=0xff;
}

int main(int argc, char **argv)
{struct sigaction sa;
 struct itimerval timer;
 if (argc>1)
    GPIO=atoi(argv[1]);
 else
    GPIO=10;
 h=red_gpio_init(GPIO);
 red_gpio_set_cfgpin(h,GPIO); // LED0

 memset(&sa, 0,sizeof(sa));
 sa.sa_handler = &test;
 sigaction(SIGVTALRM,&sa, NULL);
 /* Configure the timer to expire after TIMESLEEP msec ... */
 timer.it_value.tv_sec = 0;
 timer.it_value.tv_usec = TIMESLEEP;
 /* ... and every TIMESLEEP usec after that. */
 timer.it_interval.tv_sec = 0;
 timer.it_interval.tv_usec = TIMESLEEP;
 setitimer(ITIMER_VIRTUAL, &timer, NULL);
 printf("pointer: %p\n",h);fflush(stdout);
 while(1) {};
 red_gpio_cleanup();
 return 0;
}
```


Period

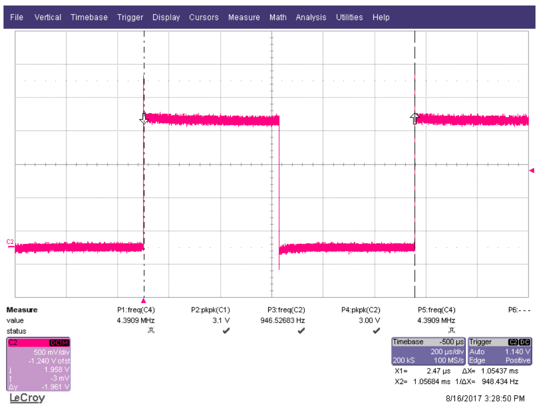

Loaded


Unloaded

The result is catastrophic: no only is sigalarm unable to meet latency constraints, but even when unloaded periodic delay jumps are observed. This delay method is unable to cope with processor load.

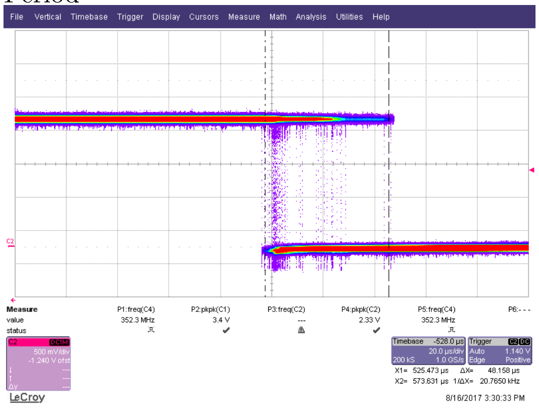
8.3 Delay using `sleep`, with Xenomai

```c
#include <signal.h>
#include <stdio.h>
#include <time.h> // nanosleep
#include <native/task.h>
#include "ledlib.h"

RT_TASK blink_task;
unsigned char *h;

void catch_signal(int sig){}

void blink(void *arg){
 int status=0;
 struct timespec tim = {0,TIMESLEEP*1000};

 rt_printf("blink task\n");
 while(1)
   {red_gpio_output(h,0,status);
    status^=0xff;
    // rt_printf("%d\n",status);
    if (nanosleep(&tim,NULL) != 0)
       {printf("erreur usleep\n");return;}
   }
}

int main(int argc, char *argv[])
{
 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0
 // Avoid memory swapping for this program
 mlockall(MCL_CURRENT|MCL_FUTURE);

 signal(SIGTERM,catch_signal);
 signal(SIGINT, catch_signal);

 rt_printf("hello RT world\n");
 rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
 rt_task_start(&blink_task, &blink, NULL);

 pause();

 rt_task_delete(&blink_task);
 red_gpio_cleanup();
 return 0;
}
```


File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
C2
LeCroy

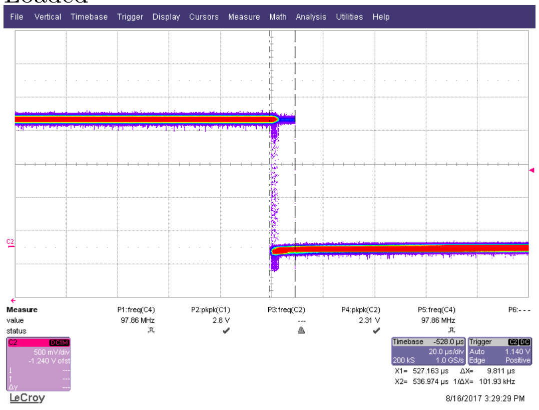
```
Measure  P1:freq(C4)  P2:pkpk(C1)  P3:freq(C2)   P4:pkpk(C2)  P5:freq(C4)  P6:- - -
value    4.3909 MHz   3.1 V        946.52683 Hz  3.00 V       4.3909 MHz
C2 DC1M  500 mV/div  -1.240 V ofst  ↓ 1.958 V  ↑ -3 mV  Δy -1.961 V
Timebase -500 µs  200 µs/div  200 kS  100 MS/s
Trigger C2 DC  Auto 1.140 V  Edge Positive
X1=  2.47 µs       ΔX=  1.05437 ms
X2=  1.05684 ms  1/ΔX=  948.434 Hz
8/16/2017 3:28:50 PM
```

Period


File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
C2
LeCroy

```
Measure  P1:freq(C4)  P2:pkpk(C1)  P3:freq(C2)  P4:pkpk(C2)  P5:freq(C4)  P6:- - -
value    352.3 MHz    3.4 V        ---          2.33 V       352.3 MHz
C2 DC1M  500 mV/div  -1.240 V ofst  ↓ ---  ↑ ---  Δy ---
Timebase -528.0 µs  20.0 µs/div  200 kS  1.0 GS/s
Trigger C2 DC  Auto 1.140 V  Edge Positive
X1=  525.473 µs    ΔX=  48.158 µs
X2=  573.631 µs  1/ΔX=  20.7650 kHz
8/16/2017 3:30:33 PM
```

Loaded

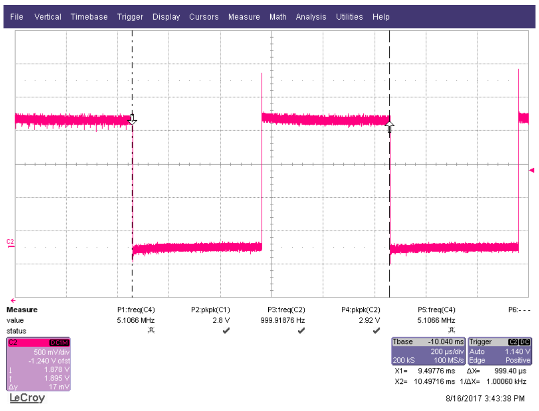

File Vertical Timebase Trigger Display Cursors Measure Math Analysis Utilities Help
C2
LeCroy

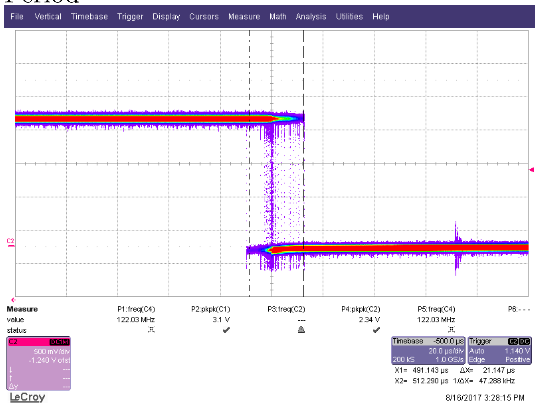
```
Measure  P1:freq(C4)  P2:pkpk(C1)  P3:freq(C2)  P4:pkpk(C2)  P5:freq(C4)  P6:- - -
value    97.86 MHz    2.8 V        ---          2.31 V       97.86 MHz
C2 DC1M  500 mV/div  -1.240 V ofst  ↓ ---  ↑ ---  Δy ---
Timebase -528.0 µs  20.0 µs/div  200 kS  1.0 GS/s
Trigger C2 DC  Auto 1.140 V  Edge Positive
X1=  527.163 µs    ΔX=  9.811 µs
X2=  536.974 µs  1/ΔX=  101.93 kHz
8/16/2017 3:29:29 PM
```

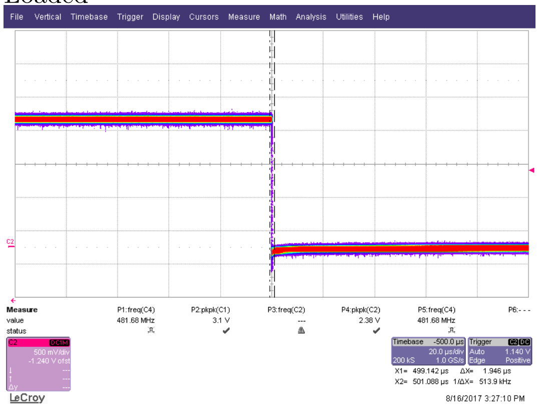
Unloaded

8.4 Delay using a timer, with Xenomai

```c
#include <signal.h>
#include <native/task.h>
#include "ledlib.h"

RT_TASK blink_task;
unsigned char *h;

void catch_signal(int sig){}

void blink(void *arg){
 int status=0;

 rt_printf("blink task\n");
 rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP*1000);
 while(1)
   {rt_task_wait_period(NULL);
    red_gpio_output(h,0,status);
    status^=0xff;
   }
}

int main(int argc, char *argv[])
{
 h=red_gpio_init();
 red_gpio_set_cfgpin(h,0); // LED0
 // Avoid memory swapping for this program
 mlockall(MCL_CURRENT|MCL_FUTURE);

 signal(SIGTERM,catch_signal);
 signal(SIGINT, catch_signal);

 rt_printf("hello RT world\n");
 rt_task_create(&blink_task, "blinkLed", 0, 99, 0);
 rt_task_start(&blink_task, &blink, NULL);

 pause();

 rt_task_delete(&blink_task);
 red_gpio_cleanup();
 return 0;
}
```


Period


Loaded


Unloaded

9 FPGA configuration

While a classical FPGA requires a dedicated framework for configuration (e.g. JTAG probe), the Zynq allows accessined the Programmable Logic part (PL) of the chip from its general purpose Processing System (PS). A framework called FPGA Manager has been standardized. The benefit of this framework is that it applies to *devicetree overlays* aimed a synchronizing all resources needed by a bitstream used for configuring the FPGA as well as the associated Linux driver. Using the devicetree overlay, the kernel module with the driver is automatically loaded when the associated bitstream defined in the plugin is requested, creating the link between the FPGA, the kernel module running on the processing system, and the userspace Linux.

9.1 Generating the encrypted bitstream

The default behavious of Vivado is to generate a bistream file with the `.bit` extension. The FPGA Manager expects another format with the appropriate header. Converting from `.bit` to `.bit.bin` is achieved with the `bootgen` tool provided with the Vivado SDK.

This tool expects a configuration file (called here `filename.bif`) with the `.bif` extension as simple as:

```
all:
{
  bitstream_name.bit
}
```

which is provided as argument to `bootgen`:

```
$VIVADO_SDK/bin/bootgen -image filename.bif -arch zynq -process_bitstream bin
```

Following the execution of this command, a file named `bitstream_name.bit.bin` has been created in the current directory.

9.2 Configuring the FPGA through `fpga_manager`

The `.bit.bin` file must be copied/moved to the `/lib/firmware` directory.

In order to tell the driver that the `PL` must be configured and which bitstream is to be used, the following command is used:

```
echo "bitstream_name.bit.bin" > /sys/class/fpga_manager/fpga0/firmware
```

leading to the display of the line

```
fpga_manager fpga0: writing nom_du_bitstream.bit.bin to Xilinx Zynq FPGA Manager
```

in the console or the kernel log, and the LED connected to `Prog done` must switch on (blue LED on the Red Pitaya).

9.3 Using a devicetree overlay for configuring the FPGA

As was the case with the previous solution, the bitstream must be located in `/lib/firmware`.

The benefit of this method over the previous one is that when the design includes IPs communicating with the processor, the *devicetree* tells the kernel which drivers to load. This driver is provided as a kernel module to be inserted when the bitstream is used to configure the FPGA. The overlay overloads the `fpga_full` node definition by providing simultaneously the bistream name as well as the identifier of the dedicated driver to be loaded for this application.

Without getting in all the details of the overlay format, the following example will allow to:

- modify the `fpga_full` node definition, `target` attribute, telling `fpga_manager` the name of the binary file to be used for configuring the PL through the `firmware-name` attribute;
- add a node defining the drivers needed to control the PL from the PS leading to the module being loaded. In the current example the associated driver is called `gpio_ctl` (`compatible` field), and the base address for sharing information between PS and PL (0x43C00000) is provided, as well as the size (0x1f) through the `reg` attribute.

```
/dts-v1/;
/plugin/;
/ {
    compatible = "xlnx,zynq-7000";
    fragment@0 {
        target = <&fpga_full>;
        #address-cells = <1>;
        #size-cells = <1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;

            firmware-name = "top_redpitaya_axi_gpio_ctl.bin";

            gpio1: gpio@43C00000 {
                compatible = "gpio_ctl";
                reg = <0x43C00000 0x0001f>;
                gpio-controller;
                #gpio-cells = <1>;
                ngpio= <8>;
            };
        };
    };
};
```

This file must be compiled using the following command:

```
/somewhere/buildroot/output/host/usr/bin/dtc -@ -I dts -O dtb -o ${FILENAME}.dtbo ${FILENAME}.dts
```

with:

- `-@` to generate symbols to be dynamically loaded when inserting the module
- `-I dts` to define the kind of input file (DeviceTree Source);
- `-O dtb` to define the kind of output file (DevideTree Binary);
- `-o` the output filename.

Loading the file into the memory requires two steps:

1. creating a directory that will hold the overlay:
   ```
   mkdir /sys/kernel/config/device-tree/overlays/toto
   ```
   will automaticall create a whole set of files in this directory:
   ```
   redpitaya> ls -l /sys/kernel/config/device-tree/overlays/toto/
   total 0
   -rw-r--r--    1 root     root             0 Jan  1 00:04 dtbo
   -rw-r--r--    1 root     root          4096 Jan  1 00:04 path
   -r--r--r--    1 root     root          4096 Jan  1 00:04 status
   ```
2. loading the overlay in the *devicetree*:
   ```
   cat gpio_red.dtbo > /sys/kernel/config/device-tree/overlays/toto/dtbo
   ```
   will configure the FPGA by transfering the bitstram, load the driver by inserting the associated module identified through the "compatible" argument which must match the corresponding field in the kernel module.

The modifications induced by the overlay can be canceled to return to the initial devicetree state by deleting the directory holding the overlay:

```
rmdir /sys/kernel/config/device-tree/overlays/toto
```

10 PS-PL interaction

10.1 GPIO

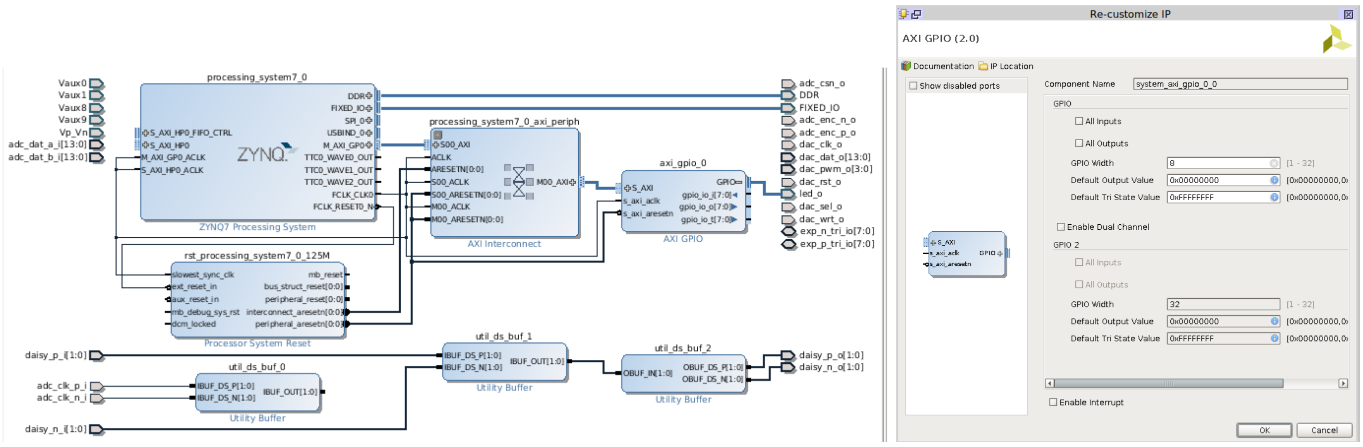
The major benefit of the Zynq architecture is to combine a general purpose Processing System (PS) processor executing GNU/Linux, and a reconfigurable Programmable Logic (PL) FPGA. We will demonstrate in the following example how to synthesize the GPIO IP configuring the PL, unlike the MIO we used earlier that can be accessed only by the PS, and how to access this new peripheral using the techniques developed in this document:

- direct access from userspace
- direct access from kernel space (module)
- accessing from a kernel driver probed by the *devicetree* using the `compatible` method.

Configuring the PL from Vivado is beyond the scope of this document. We here use a GPIO block provided by Xilinx as a Vivado IP, and connect to the PS through the AXI bus, a communication standard used on all ARM-processor based platforms (Fig. 7).

As any peripheral on digital systems, the GPIO is connected through the three interfaces of the AXI bus, namely the data bus, control bus and address bus. The logic of handling the control bus, quite complex, is hidden with the AXI Interconnect and the only information we will be interested in is the base address allocated to the IP and the address offset of each control register. The base address, varying with the kind and number of peripherals implemented in the FPGA, is provided by Vivado (Fig. 8). The address offsets to the various register holding the configuration functions are described in the GPIO bloc documentation provided by Xilinx [15]. Hence, we know how to access each PL peripheral function from the PS as we would have done on a microcontroller (Fig. 8).

[15]p.10 of `https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf`


Re-customize IP

AXI GPIO (2.0)

Component Name: system_axi_gpio_0_0

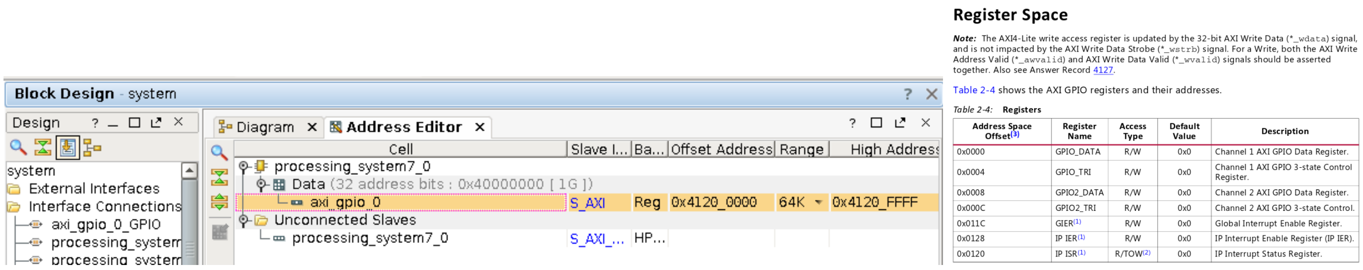
GPIO
- [ ] All Inputs
- [ ] All Outputs
- GPIO Width: 8 [1 - 32]
- Default Output Value: 0x00000000
- Default Tri State Value: 0xFFFFFFFF

- [ ] Enable Dual Channel

GPIO 2
- [ ] All Inputs
- [ ] All Outputs
- GPIO Width: 32 [1 - 32]
- Default Output Value: 0x00000000
- Default Tri State Value: 0xFFFFFFFF

- [ ] Enable Interrupt

Figure 7: The PL configured with an AXI Interconnect to link the GPIO IP in the FPGA with the PS.


Block Design - system

Cell	Slave I...	Ba...	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
Unconnected Slaves					
processing_system7_0	S_AXI_...	HP...			

Register Space

Note: The AXI4-Lite write access register is updated by the 32-bit AXI Write Data (`*_wdata`) signal, and is not impacted by the AXI Write Data Strobe (`*_wstrb`) signal. For a Write, both the AXI Write Address Valid (`*_awvalid`) and AXI Write Data Valid (`*_wvalid`) signals should be asserted together. Also see Answer Record 4127.

Table 2-4 shows the AXI GPIO registers and their addresses.

Table 2-4: **Registers**

Address Space Offset[3]	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER[1]	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER[1]	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR[1]	R/TOW[2]	0x0	IP Interrupt Status Register.

Figure 8: Address assignment of the peripherals added to the PL.

10.2 XADC

The Zynq is fitted with a slow analog to digital converter. his peripheral can be accessed either from the PS, or from the PL. Xilinx provides an IIO compatible driver as expected for such a peripheral. Without dedicated configuration of the PL, only the temperature can be read this way. We here wish to extend such PL functionalitites by updating the PL configuration to configure two pins for analog voltage measurements.

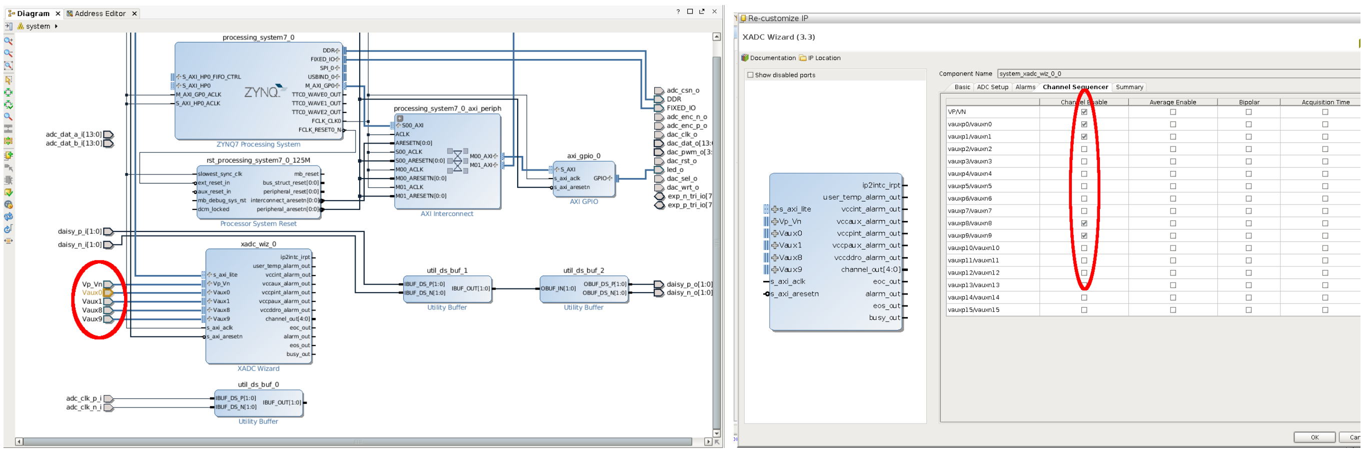
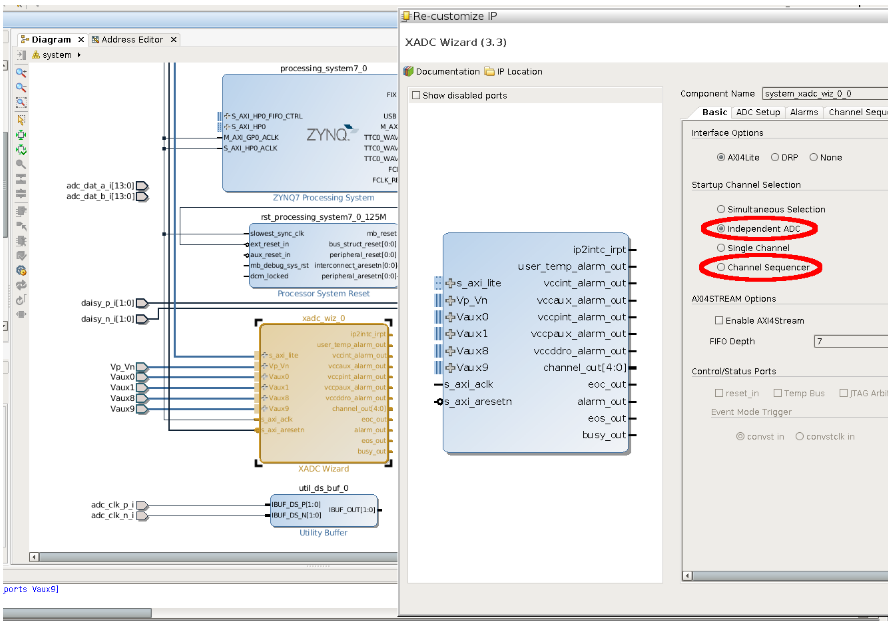


Figure 9: Steps for configuring the XADC and connect analog inputs. Connect the missing pins and run the AXI bus autorouting after having added the XADC processing block.

The devicetree overlay for communicating with this new configuration of the XADC is

```
/dts-v1/;
/plugin/;
/ {compatible = "xlnx,zynq-7000";

   fragment@0 {
        target = <&fpga_full>;
```



Component Name system_xadc_wiz_0_0

Interface Options: [✓ AXI4Lite] DRP None

Startup Channel Selection: Simultaneous Selection [✓ Independent ADC] Single Channel Channel Sequencer

AXI4STREAM Options: Enable AXI4Stream

FIFO Depth 7

Control/Status Ports: reset_in Temp Bus JTAG Arbi

Event Mode Trigger: convst in convstclk in

Figure 10: Steps for configuring the XADC and connect analog inputs. A bug in Vivado requires activating the Channel Sequencer mode before setting the Independent ADC mode when configuring the list of active channels.

```
        #address-cells = <1>;
 #size-cells = <1>;
 __overlay__ {
  #address-cells = <1>;
  #size-cells = <1>;
  firmware-name = "system_wrapper.bit.bin";
  };
 };
 fragment@1 {
  target = <&adc>;
  __overlay__ {
   xlnx,channels {
    #address-cells = <1>;
    #size-cells = <0>;
    channel@0 {reg = <0>;};
    channel@1 {reg = <1>;};
    channel@2 {reg = <2>;};
    channel@9 {reg = <9>;};
    channel@10 {reg = <10>;};
   };
  };
 };
};
```

which overloads the `&adc` entry already present in the Red Pitaya devicetree. The organization and options of the Xilinx driver for the ADC are descibed in the kernel documentation at `Documentation/devcietree/bindings/iio/adc/xilinx-xac` We indeed find in `arch/arm/boot/dts/xilinx/zynq-7000.dtsi` of the kernel sources the configuration of a Zynq-7000 with its ADC entry:

```
adc: adc@f8007100 {
   compatible = "xlnx,zynq-xadc-1.00.a";
   reg = <0xf8007100 0x20>;
   interrupts = <0 7 4>;
   interrupt-parent = <&intc>;
   clocks = <&clkc 12>;
};
```

which is overloaded with the new configuration. The Xilinx driver, defined as compatible with `xlnx,zynq-xadc-1.00.a`, is configured to comply with the IIO inputs and outputs as described in the documentation cited previously. We hence just need to modify the devicetree, without having to update the driver code to access the new resources implemented in the bitstream including the XADC connected to the PS through the AXI bus. We check the location of the driver module with

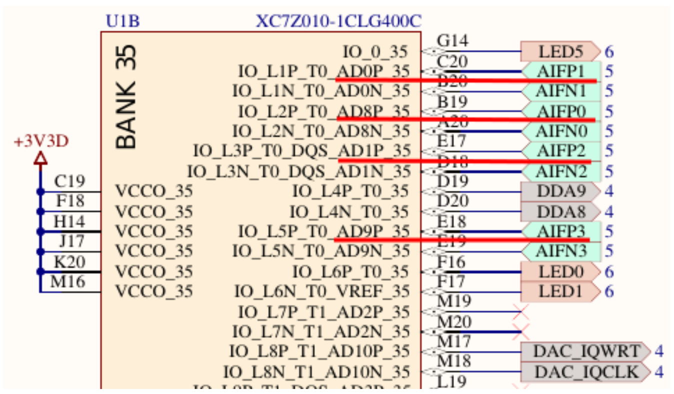
```
$ LINUX/drivers/iio/adc$ grep xlnx * | grep adc
xilinx-xadc-core.c:     { .compatible = "xlnx,zynq-xadc-1.00.a", (void *)&xadc_zynq_ops },
```

which is indeed stored with other IIO drivers.

The base address of the XADC connected to the PS is documented at page 1160 of `ug585-Zynq-7000.pdf`: the registers for configuring the XADC from the PS are located at addresses starting at 0xf8007100. In order to comply with the XADC address organization in the bitstream et describe this organization in the devicetree when loading the IIO driver, we must compile the `xilinx_xadc.ko` driver as a module (as opposed to a static link into the kernel). Once loaded, the IIO peripheral is accessible at `/sys/bus/iio/devices/iio:device0`.

The `Red_Pitaya_Schematics_STEM_125-10_V1.0.pdf` schematic describes the relation between the information screen printed on the printed circuit board and the Zynq pin names and their functions: AI0 is connected to AD8, AI1 is connected to AD0, AI2 is connected to AD1 and AI3 is connected to AD9, which helps understanding the naming chosen in the IIO driver:

- pin 16 E2=in_voltage8_vaux9_raw
- pin 13 E2=in_voltage9_vaux8_raw
- pin 14 E2=in_voltage11_vaux0_raw
- pin 15 E2=in_voltage10_vaux1_raw



In order to access the XADC registers from the PL, their address is described in the documentation of the XADC Wizard v3.0 (reference PG091, version April 1, 2015): for example the register for accessing ADC8 is located at offset 0x260 with respect to the base address provided by Vivado.

A Sending commands through a TCP/IP server

A basic example only allowing for a single connection (no multi-threading) of a TCP/IP server is as follows:

```c
#include <sys/socket.h>
#include <resolv.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>

#define MY_PORT         9999
#define MAXBUF          1024

int main()
{int sockfd;
 struct sockaddr_in self;
 char buffer[MAXBUF];

 sockfd = socket(AF_INET, SOCK_STREAM, 0);  // ICI LE TYPE DE SOCKET

 bzero(&self, sizeof(self));
 self.sin_family = AF_INET;
 self.sin_port = htons(MY_PORT);
 self.sin_addr.s_addr = INADDR_ANY;

 bind(sockfd, (struct sockaddr*)&self, sizeof(self));
 listen(sockfd, 20);

 while (1)
 {struct sockaddr_in client_addr;
  int taille,clientfd;
  unsigned int addrlen=sizeof(client_addr);

  clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
  printf("%s:%d connected\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
  taille=recv(clientfd, buffer, MAXBUF, 0);
  send(clientfd, buffer, taille, 0);
  close(clientfd);
 }
 close(sockfd);return(0);  // Clean up (should never get here)
}
```

Reaching this server is for example achieved by running `telnet IP 9999` for connecting to port 9999 of the IP address of the server, or thanks to `netcat` with `echo "message | nc IP port`.

B Modifying the Buildroot configuration

Standard tools needed for the lab sessions:

User space	(`make menuconfig`)
screen	
boa	
file	
stress	
vim	(after activating *show packages that are also provided by busybox*)
nano	

Updates to the kernel configuration to allow accessing the GPIOs and LEDs, as well as configuring the FPGA:

Kernel space	(`make linux-menuconfig`)
LED driver as module	`Device Drivers` → `LED Support` → `<M> LED Class Support` et `<M> LED Support for GPIO connected LED`
GPIO driver as module	`Device Drivers` → `GPIO Support` → `Memory mapped GPIO drivers` → `<M> Xilinx GPIO support` et `<M> Xilinx Zynq GPIO support`
xdevcfg as module	`Device Drivers` → `Character devices` → `Xilinx Device Configuration`
FPGA Manager as module	`Device Drivers` → `FPGA Configuration Support` → `<M> FPGA Configuration Framework`

References

[1] P. Ficheux, *Les distributions "embarquées" pour Rasberry PI*, Opensilicium 7 (2013)

[2] P. Kadionik, P.Ficheux, *Temps réel sous LINUX (reloaded)*, GNU/Linux Magazine France HS 24 (2006), disponible à `http://pficheux.free.fr/articles/lmf/hs24/realtime/linux_realtime_reloaded_final_www.pdf` et `http://www.unixgarden.com/index.php/gnu-linux-magazine-hs/temps-reel-sous-linux-reloaded`

[3] J. Corbet, A. Rubini, & G. Kroah-Hartman, *Linux Device Drivers, 3rd Ed.*, O'Reilly, disponible à `http://lwn.net/Kernel/LDD3/`

[4] *Zynq-7000 All Programmable SoC Technical Reference Manual, rev. 6 Dec. 2017*, Xilinx (2017)

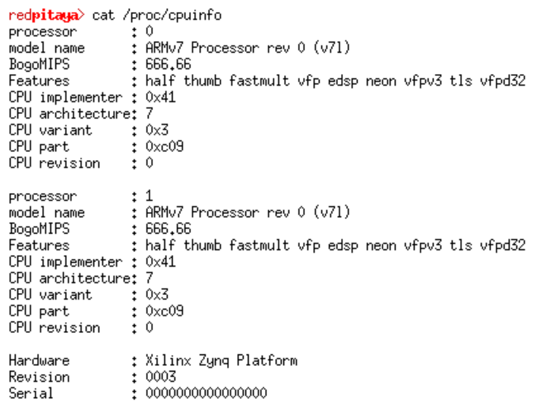
[5] C. Blaess, *Interactions entre espace utilisateur, noyau et matériel*, GNU/Linux Magazine France Hors Série 87 (Nov.–Déc. 2016)

[6] C. Blaess, *Solutions temps réel sous Linux*, Eyrolles (2012)

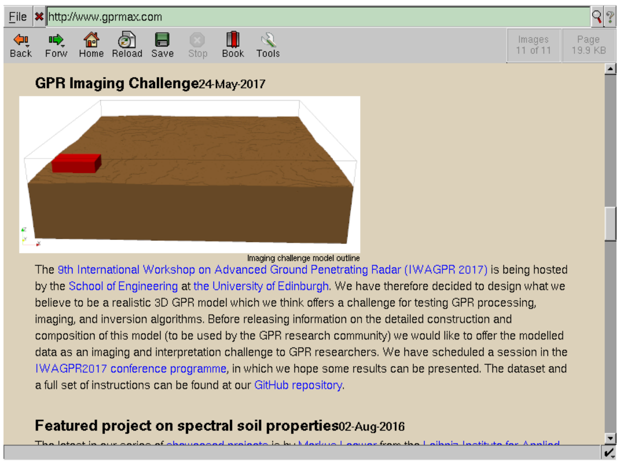
Activating the X11 server

The Red Pitaya is not fitted with a graphical interface port. Nevertheless, graphical applications can be execued, thanks to a X11 server running on the Red Pitaya and displaying on the client running on the PC (X11 swaps the usual convention of server and client naming convention).

Activating the graphical interface capability is achieved by running in Buildroot the `X.org` menu in `Target Packages` → `Graphic libraries and applications (graphic/text)`. Remember to install `xauth` in order to handle certificates allowing for exporting the graphical interface from the Red Pitaya to the host PC.



```
redpitaya> cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 666.66
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

processor       : 1
model name      : ARMv7 Processor rev 0 (v7l)
BogoMIPS        : 666.66
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc09
CPU revision    : 0

Hardware        : Xilinx Zynq Platform
Revision        : 0003
Serial          : 0000000000000000
```



GPR Imaging Challenge24-May-2017

Imaging challenge model outline

The 9th International Workshop on Advanced Ground Penetrating Radar (IWAGPR 2017) is being hosted by the School of Engineering at the University of Edinburgh. We have therefore decided to design what we believe to be a realistic 3D GPR model which we think offers a challenge for testing GPR processing, imaging, and inversion algorithms. Before releasing information on the detailed construction and composition of this model (to be used by the GPR research community) we would like to offer the modelled data as an imaging and interpretation challenge to GPR researchers. We have scheduled a session in the IWAGPR2017 conference programme, in which we hope some results can be presented. The dataset and a full set of instructions can be found at our GitHub repository.

Featured project on spectral soil properties02-Aug-2016

Figure 11: Left: Xterm executed on the Red Pitaya and displayed on the host PC. Right: Dillo allows for displaying HTML web pages from the Red Pitaya.

.1 Qt5 graphical user Interface

Qt5 is a set of multiplatform C++ libraries compatible with multiple operating systems (MS-Windows, X11 especially on GNU/Linux, OS X or iOS for Apple) and hardware platforms (x86, MIPS, ARM, SPARC ...). Despite is heavy burden on computational requirements as is the case for all graphical interface, it emphasizes the potential benefits of running an operating system on the embedded system.

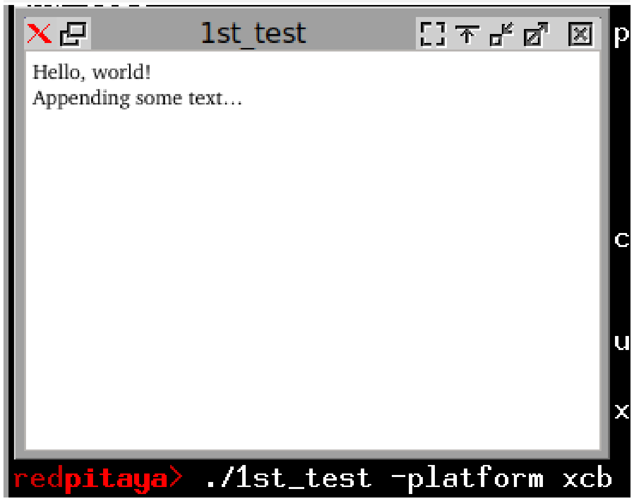
Compiling Qt5 support on Buildroot requires C++ support, which is not active in the default configuration. We make sure the whole toolchain is re-compiled with C++ support by deleting the `output` (`rm -rf output`) directory in the Buildroot archive and running again `make`.

Furthermore, Qt5 fonts are not included by default, and requires calling (`fontconfig`) whose computational requirements are too heavy for an embedded system. The most efficient approach is to include static fonts by activating the `liberation` package in the `Target packages` → `Fonts, cursors, icons, sounds and themes` menu and, on the Red Pitaya, to create on the embedded platform the symbolic link between the location in which fonts are stored, (`/usr/share/fonts/liberation`), and the directory in which Qt5 expects the fonts to be located (`/usr/lib/fonts`):

```
cd /usr/lib
ln -s /usr/share/fonts/liberation/ fonts
```

Once the system has been installed, we can develop and compile a basic test application. The trivial program proposed (initially for Qt4, but also functional with Qt5) at `http://doc.qt.io/qt-4.8/gettingstartedqt.html` is saved in a filename with a `.cpp` extension (Fig. 12).

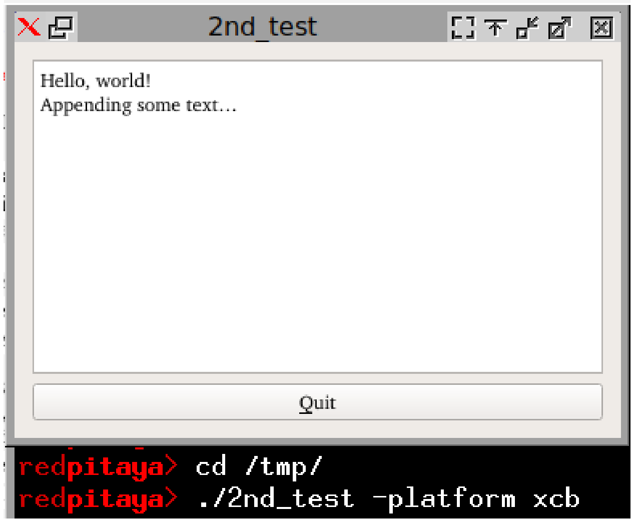
```
// https://wiki.qt.io/How_to_Use_QTextEdit

#include <QApplication>
#include <QTextEdit>

int main(int argc, char **argv)
{
 QApplication app(argc, argv);

QTextEdit *txt = new QTextEdit();
 txt->setText("Hello, world!");
 txt->append("Appending some text...");

txt->show();
 return app.exec();
}
```



```
Hello, world!
Appending some text...
```

```
redpitaya> ./1st_test -platform xcb
```

```
// http://doc.qt.io/qt-4.8/gettingstartedqt.html

#include <QApplication>
#include <QTextEdit>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argv, char **args)
{
 QApplication app(argv, args);

 QTextEdit *textEdit = new QTextEdit;
 textEdit->setText("Hello, world!");
 textEdit->append("Appending some text...");

 QPushButton *quitButton = new QPushButton("&Quit");
 QObject::connect(quitButton, SIGNAL(clicked()), qApp, SLOT(quit()));

 QVBoxLayout *layout = new QVBoxLayout;
 layout->addWidget(textEdit);
 layout->addWidget(quitButton);

 QWidget window;
 window.setLayout(layout);
 window.show();

 return app.exec();
}
```



```
Hello, world!
Appending some text...
```

```
redpitaya> cd /tmp/
redpitaya> ./2nd_test -platform xcb
```

Figure 12: Left: example of a text editor, one of the widgets provided by the Qt5 library. Right: a slightly more advanced example with buttons.

⚠ The extension of the filename is important: a `.c` extension results in an attempt to compile using `gcc` which does not understand C++ syntax nor knows how to link with the appropriate libraries.

Qt provides the means for generating dynamically the `Makefile` from a configuration file whose extension is `.pro`. This file is automatically generated by running `qmake -project`. However, we must make sure we run `qmake` from the `output/host/usr/bin` directory of Buildroot and *not the qmake possibly installed on the host (PC)*. Having generated the `.pro` configuration file, we add the Qt modules required by the proposed program, in our case the widgets module, with

```
QT += widgets
```

to finally have a configuration file looking like

```
TEMPLATE = app
TARGET = Qt
INCLUDEPATH += .

QT += widgets

# Input
SOURCES += 2nd_test.cpp
```

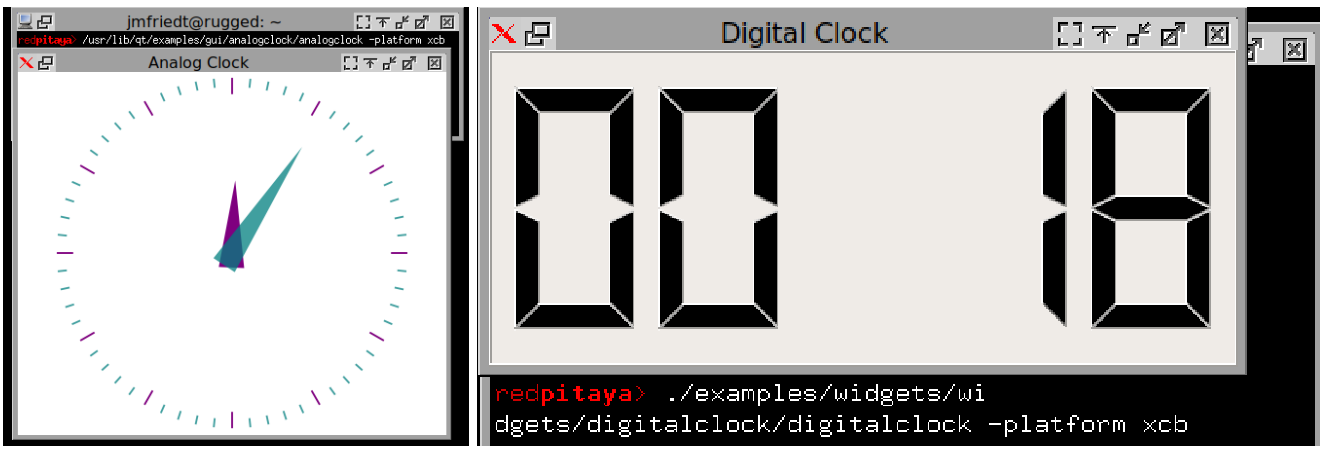
We run again `$BR/output/host/usr/bin/qmake` – this time without the `-project` option – to generate the Makefile, and finally the source code is compiled to a binary executable with `make`.

Once the compilation has been completed, two results are obtained:

- the GNU/Linux image (rootfs) for the ARM target including the Qt libraries,orte les
- the host (PC) now has the Qt tools for configuring and cross-compiling Qt5 programs written in C++ and calling Qt libraries.

The program is executed on the Red Pitaya platform by specifying the graphical output, here *xcb*: after transfering the binary excecutable to the Red Pitaya target platform, we execute `Qt -platform xcb` and, if all goes well, a window with a text editor should be displayed (Fig. 12, bottom)

Once the procedure for compiling a program has been understood, the Qt langage and libraries are documented with many examples given in the archive at `$BR/output/build/qt5base-5.*/examples`. We have hence checked that the exemples `gui/analogclock` or `widgets/digitalclock` are working properly (Fig. 13).



```
redpitaya> /usr/lib/qt/examples/gui/analogclock/analogclock -platform xcb
```

```
redpitaya> ./examples/widgets/wi
dgets/digitalclock/digitalclock -platform xcb
```

Figure 13: Left: the `analogclock` example. Right: the `digitalclock` example. Both examples are found in `/usr/lib/qt/examples/`, and are executed on the `xcb` plarform specified with the `-plateform` argument when running the executable.